

INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN



Master's Thesis

Fragment Based Heuristics for Level-Of-Detail Selection

Lara Christina Hirschbeck



Master's Thesis

Fragment Based Heuristics for Level-Of-Detail Selection

Lara Christina Hirschbeck

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller
Betreuer: MNM-Team-Betreuer Dr. Christoph Anthes
MNM-Team-Betreuer Markus Wiedemann
Abgabetermin: 21. August 2017

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 18. August 2017

.....
(Unterschrift des Kandidaten)

Abstract

I present an extension for an existing 3D framework, which provides additional features for level-of-detail selection for objects in a 3D scene at run-time.

I implement and analyse an existing algorithm for determining the level of detail for objects within a 3D scene. This algorithm is based on a cost-benefit calculation. I propose some practical implementation variants and adjustments for determining the cost and benefit of the various objects within the scene.

This adjustments are first, another formula for the cost equation and second, three variations for determining the cost and benefit of the scene depending on pixels and fragments.

To examine the differences between the calculations, I use a sample scene with a frame-mapped camera move through the scenery. To get a reference, the scene is rendered on the highest resolution for all objects and on the lowest resolution for all objects.

The adjustment to the cost formula seems to be slightly better, as it does not create as much overhead in rendering time, as the original formula. The variations for determining the cost on the other hand need further work.

Zusammenfassung

Im Rahmen dieser Masterarbeit wurde eine Erweiterung für eine bestehende Programm-Bibliothek entwickelt. Diese ermöglicht es einer Anwendung, die diese Bibliothek verwendet, verschiedene Auflösungen eines Objekts zur Laufzeit, je nach Bedarf, zu laden.

Diese Masterarbeit untersucht einen gegebenen Algorithmus zur Bestimmung des zu verwendenden Detailgrads eines Objektes in einer 3D Szene. Im Zentrum dieses Algorithmus steht eine Kosten-Nutzen-Rechnung. Daher werden einige Methoden zur konkreten Implementierung des Algorithmus in dieser Arbeit vorgestellt. Es wird auch eine leichte Abänderung der verwendeten Kosten-Formel vorgeschlagen.

Die Anpassung beinhaltet drei Varianten um die Kosten und Nutzen auf Basis von Pixeln und Fragmenten zu berechnen. Um die Unterschiede der Berechnungsmethoden vergleichen zu können, wird eine Kamerafahrt verwendet, die fest an die Einzelbilder gekoppelt ist.

Um eine Referenz für die Berechnungsdauer zu erhalten, wird die Szene zusätzlich einmal für alle Objekte auf der höchsten und einmal auf der niedrigsten Auflösung durchgerechnet.

Im Vergleich mit der Original-Formel erscheint die vorgeschlagene Abänderung leichte Vorteile zu bieten, da sich hierdurch weniger zusätzliche Berechnungszeit ergibt. Die Varianten für die Berechnung der Pixel und Fragmente dagegen benötigen weitere Anpassungen.

Contents

1. Introduction	1
2. Fundamentals and Related Work	3
2.1. Fundamentals	3
2.1.1. Rendering pipeline	3
2.1.2. Fragment shader	4
2.1.3. Level of detail	5
2.2. Related Work	6
2.2.1. Funkhouser and Séquin’s algorithm	7
2.2.2. Work based on Funkhouser and Séquin	10
3. Concept	15
3.1. Previous considerations	15
3.2. Integration of Funkhouser and Séquin’s algorithm	15
3.3. Adaption of Funkhouser and Séquin’s cost formula	16
3.4. Estimation of fragments	17
3.5. Estimation of pixels	18
3.5.1. Lookup table	18
3.5.2. Bounding box	18
3.6. Visualisation	18
3.6.1. Colour scheme	19
3.6.2. Colour mapping	20
4. Implementation	21
4.1. Included libraries and environment	21
4.2. Implementation of Funkhouser’s algorithm	21
4.2.1. Determining coefficients	21
4.2.2. Funkhouser’s advanced algorithm	22
4.3. Size estimation features	26
4.3.1. Estimation via bounding box	26
4.3.2. Preprocessing and lookup tables	27
4.4. Program arguments and control keys	31
5. Results and Discussion	33
5.1. Sphere	33
5.2. Test scene	33
5.2.1. Scene components	33
5.2.2. Camera simulation	34
5.2.3. Estimation bounding box vs. estimation lookup table vs. real pixels	35
5.2.4. Estimation bounding box vs. estimation lookup table vs. real fragments	36

Contents

5.3. Comparing cost formulas	37
5.4. Comparing graphs of fragments, pixels and costs	39
5.5. Cost estimation with pre-processing	40
5.5.1. Cost	40
5.5.2. Memory	40
6. Conclusion and Future Work	47
7. Acknowledgements	49
List of Figures	51
List of Tables	55
Bibliography	57
A. Appendix	61

1. Introduction

3D models are used in many fields, from the more entertainment oriented ones of computer games and artistic applications, on the one hand, to a whole universe of scientific applications, on the other hand.

The necessary time for rendering 3D models depends on the complexity of input data, the complexity of visualisation and the hardware used. The hardware is usually pre-defined, as is the visualisation in the form of the desired shader, which can only be changed within a small range without significantly changing the visual quality. Only the input data, more specifically the scene of 3D models, is easily mutable. The goal is to make the most of the given resources in order to provide the best visual presentation to the user, while keeping the framerate constant.

When examining pictures, objects in the background are usually less detailed than comparable objects in the foreground. This leads to the idea of using multiresolutional objects in 3D modeling. If the importance of an object increases in a certain frame, the program depicts a higher resolution for it. The importance can be estimated by the size of the object or its semantical value. If the same object loses importance later on, a lower resolution can be chosen.

Current Massively Multiplayer Online Games (MMO) like Guild Wars 2 [Are17] are a common example for the use of various levels of detail (LOD). A player represented by an avatar is moving around in the digital environment and approaches a group of other avatars, all controlled by other players. From a distance all male and female avatars are rendered with the same basic settings for hair colour, armoury and weaponry. If the avatar of the player gets closer to the group, those are rendered with more details like individual hair colour, legendary armour and special weapons. It takes time to transmit the data from the servers to the client computers of the gamers located all around the world. For a small amount of players it could be possible to store those data fixed on the respective clients. For a high number of players, as they occur in MMOs, it is more common to send the data when a certain threshold of distance is met. This makes even more sense when all users can individualise their characters all through the game. Then the look of the other avatars is updated only at client computers which need the data for rendering. But the distance is one part of an algorithm which determines the LOD rendered on the client. If the client is not a high performance machine, it can not render all the details of the object at their highest tier, even if all the data would be transferred in time. The bottleneck in this scenario is the memory and the graphic card. The user can adjust the game settings to a low- or mid-res setting, which then allows the computer to render the environment at an acceptable framerate. In an ideal program no manual adjusting by the player would be needed, since a self regulating system would choose the respective LODs on its own. In my work, I present three possibilities based on which such a program could choose the resolution of an object.

In the second chapter, I give a summary of fundamentals and related work relevant for a better understanding of some concepts used in this thesis. I explain the concept of the general algorithm and the variations I made from the original implementation. In chapter 4,

1. Introduction

I give an overview over the concrete implementation of Funkhouser and Séquin's algorithm and challenges, which arose while working on the project. This is followed by the results and discussion, where I compare the three possible size estimations to each other under different test cases and debate the particular outcomes of the findings. The last chapter is about my conclusions from this project and future work, which has potential to improve this approach.

2. Fundamentals and Related Work

In the first part of this chapter, I present background information relevant for understanding the following sections. In the second part, I describe work from other authors upon which I built this thesis.

2.1. Fundamentals

In this section, I detail basic principles which are essential to fully comprehend the subsequent chapters. I discuss the rendering pipeline of OpenGL, since two of my approaches rely on modifications to the rendering process. Furthermore, I explain the concept of level of detail (LOD), as the switching of LODs is the core principle of the algorithm my thesis is based on.

2.1.1. Rendering pipeline

Rendering objects in computer graphics needs several steps in order to convert a geometrical form into the pixels shown on a display. These steps are called *rendering pipeline* [SSK13, ch. 1]. The OpenGL pipeline has evolved and changed since its introduction. For my implementation, I use OpenGL 4.3 and its corresponding rendering pipeline (see Figure 2.1).

The first stage serves to collect *vertex data*. Then, for every vertex which will be drawn, the vertex shader is called. Its concrete tasks vary from passing through data to calculating colours and the vertex' screen position. Thereafter comes the *tessellation control shader*. If it is activated, it controls the tessellation a particular patch (a kind of geometric primitive) gets supplied. It also filters vertex data coming from the previous stage. Its main purpose is to provide the tessellation levels to the next stage. The *tessellation evaluation shader* takes the data provided by the tessellation control shader and computes the positions and further data concerning each individual vertex. In *geometry shading* additional processing of the geometry primitives can be done, similar to the tessellation shader. As the previous tessellation stage and the fragment shader in the following, it is not obligatory to use the geometry shader stage. Although, when called, it serves the purposes of layered rendering and transform feedback. The vertices on which the previous stages are based are then organised into their associated geometric primitives in the *primitive setup*. Vertices outside the viewport are clipped and rasterization is executed. The *rasterizer* creates fragments for every object within the *clipping* region, that is the area, where objects are within camera sight. Fragments are pixels-to-be, whose data still could be changed by, for example, a depth test. This takes place in the penultimate step, the *fragment shader*. The fragment shader is executed for every generated fragment. *Per-sample operations* take the output data from the fragment shader and write them to various buffers. This results in the picture rendered on the screen.

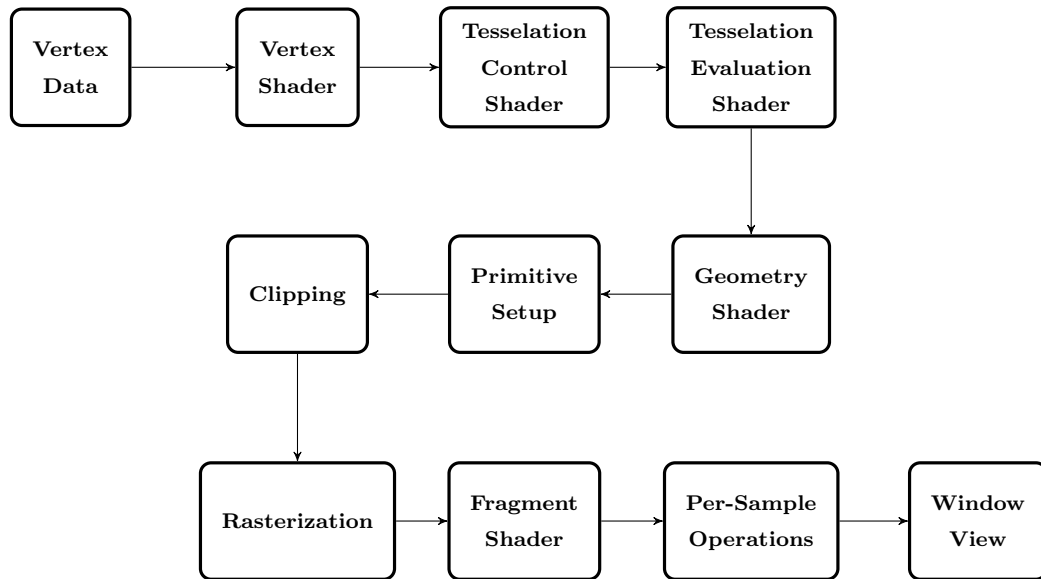


Figure 2.1.: Rendering pipeline under OpenGL 4.3 [SSKLLK13, ch. 1].

2.1.2. Fragment shader

Before showing any picture on the screen, various operations need to be executed on the fragments. Fragments work as “candidate pixels” [SSKLLK13, ch. 1]. The data for the pixels showing up on screen are already in the framebuffer, but fragments can still be changed or rejected by depth tests, for example. They include not only the respective color value and/or texture coordinates, but also further data like raster position, depth, stencil, alpha and windowID. The created fragments are not combined with each other, but with the data already present in the frame buffer. If depth test is activated and a fragment is farther away than the pixel at the corresponding location, then the fragment is discarded. If it is nearer than the existing pixel, the fragment data replaces that pixel completely. In case of alpha blending a mixture of the fragment’s and the pixel’s existing color replaces the pixel. An example for this case could be a translucent object through which another object is visible.

The *fragment shader* terminology is specific to OpenGL. Other graphic APIs use other terminology (e.g. pixel shader in DirectX). It is the final stage in the rendering pipeline (see Figure 2.1) where it is still possible to have programmatic control of any kind over the colour of a position on screen. To visualise the impact of the fragment shader on a scene, Figure 2.2 shows the Stanford Armadillo with its regular bounding box. On the left the armadillo is shown with discarded fragments lesser than 0.97 in `gl_FragCoord.z` and on the right with discarded fragments greater and equal than 0.97 in `gl_FragCoord.z`. The images provide a kind of sliced view on the 3D model showing the front on the left and the back in front view on the right. When combined, they result in the whole 3D model again. There are more fragments than pixels created and the combination of the values results in the finally displayed pixels on the screen.

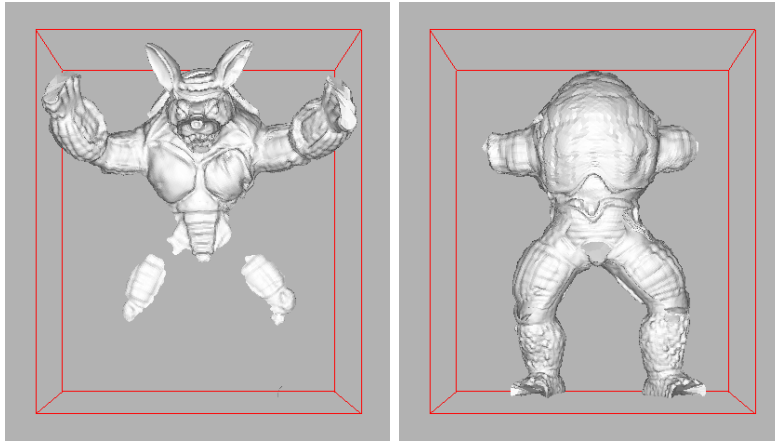


Figure 2.2.: Stanford Armadillo with bounding box in red and fragment shader configured to discard fragments lesser, respectively greater/equal than 0.97 on z-coordinate.

2.1.3. Level of detail

In 1976, Clark [Cla76] was the first who reportedly described the idea of using various resolutions within a 3D scene. He proposes hierarchical geometric models for achieving an optimal visualisation and the best use of resources like memory and processing power. He presents a hierarchical structure for increasing the range of complexity of a virtual environment. A fixed upper limit tops the visible complexity of a scene. An algorithm varies the amount of detail of each object within the scene. Since then, the general idea of a dynamic amount of detail has not changed. But new algorithms were developed for choosing the level of detail (see [FS93] and [EMB01]), the creation of the LODs (see [XESV97] and [VM02]) and the processing and managing of model data (see [AL99], [KS99] and [Lin03]).

In this subsection, I give an overview of level of detail and its applications. The general idea of LODs is to reduce the amount of vertices of an object depending on its visibility in a scene. This reduces also the rendering costs of this object. According to Peng et al. [PPCT11], there are two main classifications of LODs: Continuous LODs and discrete LODs.

Continuous LODs

The use of continuous LODs requires preprocessing of the object, as presented by Xia et al. [XESV97]. This results in hierarchical structures which contain the vertices of the object. By means of these structures, the algorithm is able to change the appearance of the object during runtime. The necessity of changing LODs depends, among others, on the object's distance to camera, its respective size on the screen or the used rendering mode. Figure 2.3 and Figure 2.4 show the gain in visual fidelity of a sphere by switching the LOD type from uniform to adaptive. The main disadvantage of continuous LODs is the increase in rendering time due to the additional calculation of the visible vertices.

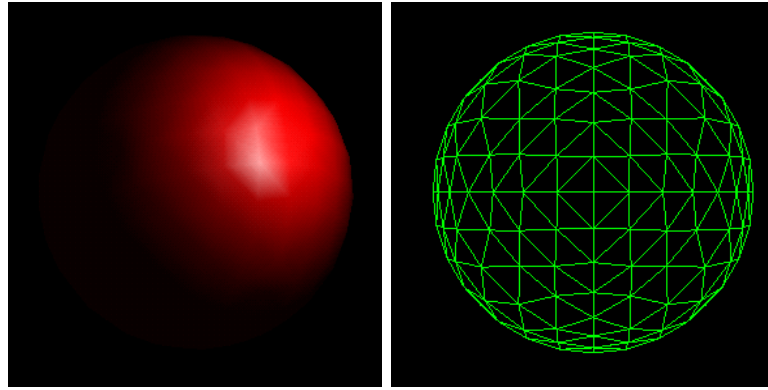


Figure 2.3.: Sphere with 512 triangles (uniform LOD) [XESV97].

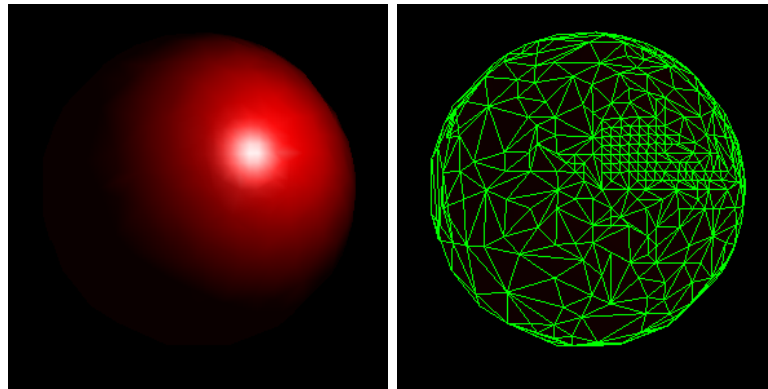


Figure 2.4.: Sphere with 537 triangles (adaptive LOD) [XESV97].

Discrete LODs

The simpler approach is to have a number of pre-rendered resolutions for each object at hand. Figure 2.5 shows the Stanford Bunny with decreasing amount of triangles from left to right. On the left, a rendering with the full resolution of 69 451 polygons is shown. The right most model features the lowest resolution with 76 triangles. The same objects are then rearranged in smaller appearance, creating a 3D perspective in Figure 2.6. Because the objects with lower resolution are also smaller, the already missing details do not stand out. The different resolutions are either created externally before the execution of the program or they can be created automatically within an initialisation step of the program. The respective LOD suitable for the scene is chosen by an algorithm, such as the one by Funkhouser and Séquin [FS93].

2.2. Related Work

In this section, I give an overview of several algorithms for choosing an appropriate LOD. The first subsection concentrates on the work of Funkhouser and Séquin and some of its variants. The second subsection is about topics by other authors, which themselves are strongly related to Funkhouser and Séquin's work.

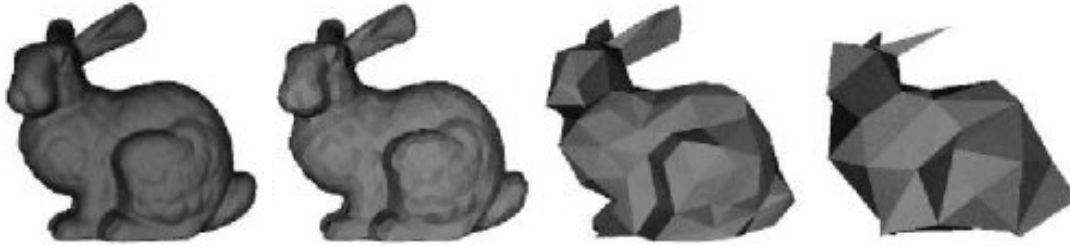


Figure 2.5.: Stanford bunny with a decreasing amount of vertices. Triangles from left to right: 69 451, 2 502, 251, 76 [Sch08].

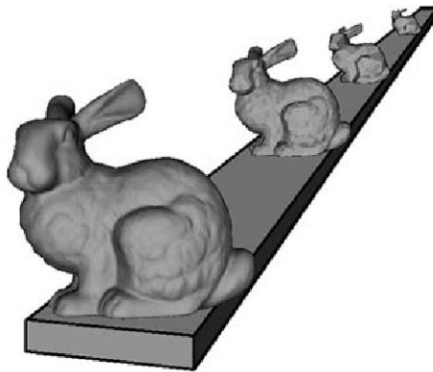


Figure 2.6.: Stanford bunny with decreasing amount of vertices and smaller appearance, creating a 3D perspective, from left to right [Sch08].

2.2.1. Funkhouser and Séquin’s algorithm

The rough basis for my work is given by Funkhouser and Séquin [FS93]. They follow the concept “do as well as possible in a given amount of time”. In the center of Funkhouser and Séquin’s algorithm is a cost-benefit calculation. The objects showing up in a scene are represented as tuples (O, L, R) , where O is an instance of an object, L is the level of detail and R stands for the used rendering algorithm. An overview for this is given in Table 2.1. Therefore, there are heuristics for both $Cost(O, L, R)$ and $Benefit(O, L, R)$.

In total, $Cost$ and $Benefit$ are used to choose the LOD for each potentially visible object. This means, that the sum shown in equation 2.1 over the benefit of all objects O with the respective level of detail L and rendering process R needs to be maximised. This sum is build with subject to the sum over the costs of all objects with the respective LOD and rendering process. The sum over the costs are kept under or equal to a maximum frame rate $TargetFrameRate$, see equation 2.2.

$$\sum_S Benefit(O, L, R) \tag{2.1}$$

$$\sum_S Cost(O, L, R) \leq TargetFrameRate. \tag{2.2}$$

2. Fundamentals and Related Work

Abbr.	meaning
O	object
L	level of detail (LOD)
R	rendering algorithm
(O, L, R)	tuple consisting of object, LOD and rendering algorithm
S	set of object tuples
C	coefficient
$TargetFrameRate$	framerate which should be achieved

Table 2.1.: Abbreviations used by Funkhouser and Séquin.

S in this case stands for a set of object tuples rendered in each frame. The cost heuristic contains C_1 , C_2 and C_3 , which are constant coefficients specific to a rendering algorithm and machine.

$$Cost(O, L, R) = \max\left(C_1 \cdot Poly(O, L) + C_2 \cdot Vert(O, L), C_3 \cdot Pix(O)\right) \quad (2.3)$$

Those coefficients are determined empirically by rendering sample objects with a wide variety of sizes and LODs. The rendering times are then plotted against the number of polygons, vertices, and pixels drawn. The slope of the best fitting line through these points is equivalent to the coefficients.

The benefit heuristic 2.4 depends primarily on the *Size* of an object tuple in the eventually drawn image. The *Size* is an estimation of the number of pixels covered by the object.

$$Benefit(O, L, R) = Size(O) \cdot Accuracy(O, L, R) \cdot Importance(O) \cdot Focus(O) \cdot Motion(O) \cdot Hysteresis(O, L, R) \quad (2.4)$$

Accuracy 2.5 describes the relation of a used LOD to the scene where all objects are rendered with their highest resolution. $Samples(L, R)$ stands for the number of pixels in a ray tracing rendering algorithm, the number of vertices, if Gouraud shading is used, or the number of polygons, if the scene is rendered with flat shading. Funkhouser states also, that the value of $Samples(L, R)$ should never exceed the respective pixel count. The exponent m is related to the error of the used interpolation method. For flat shading m is 1, for Gouraud shading it amounts to 2. The *BaseError* is set to 0.5, since it gives a strong error in case a curved surface is represented by a single flat polygon. This choice still leads to a significantly higher benefit than not rendering the surface at all.

$$Accuracy(O, L, R) = 1 - Error = 1 - \frac{BaseError}{Samples(L, R)^m} \quad (2.5)$$

Zachmann [Zac16] uses a slightly different interpretation of Funkhouser's *Accuracy* formula in his lecture slides:

$$Rendering(O, L, R) = \begin{cases} 1 - \frac{c}{pgons}, & \text{if flat} \\ 1 - \frac{c}{vert}, & \text{if Gouraud} \\ 1 - \frac{c}{vert}, & \text{if Phong} \end{cases}$$

Here c is equivalent to the *BaseError* given by Funkhouser and Séquin. The value of $pgons$, respectively $vert$, is equivalent to Funkhouser and Séquin’s $Samples(L,R)$ without the exponent m , which Zachmann discards.

Importance decreases or increases the need for rendering a specific object on a higher LOD. It represents the semantical value of the object in its context. A pencil in an adventure game, for instance, could be more important to the player than the desk on which it lies. The domain of *Importance* is $[0.0; 1.0]$. *Focus* is usually set on objects in the center of the screen. The position of the object relative to the centre leads to a multiplier between 0.0 and 1.0, too. If the object is close to the centre, the value is higher. If it is further away from the centre and closer to the edge, its value is lower. *Motion* stands for *Motion Blur* and decreases the benefit of an object. If the object is moving quickly across the scene it adopts a lower LOD, as the user’s eyes would need time to focus. Here, too, a value between 0.0 and 1.0 is used. If the speed of the object relative to the camera is higher, the value is decreased. If the speed is lower, the value is increased. *Hysteresis* is a concession to a persistent visualisation experience of the user. If an object should repeatedly be rendered with different LODs in successive frames, it can be bothering the experience of the user. When the LOD of the object is repeatedly altered, it can lead to a perception of flickering for the user. Therefore, *Benefit* is reduced by an amount proportional to the difference of LOD from the previous frame. This results in a factor between 0.0 and 1.0, as well. So, in conclusion, *Benefit* is always greater than or equal to 0 for every object, because it works as a product of the aforementioned factors.

Any implementation which solves the formulas 2.1 and 2.2, would lead to an NP-complete optimisation problem. It can be reduced to the Continuous Multiple Choice Knapsack Problem, a more specific version of the Knapsack Problem [IHTI78]. In the Knapsack Problem elements are partitioned into candidate sets. Only one element from each candidate set may be put in the knapsack at the same time. In Funkhouser and Séquin’s case, the set S of object tuples rendered is the knapsack, the object tuples are the elements to be put into the knapsack, the target frame time is the size of the knapsack and the sets of object tuples representing the same object, i.e. the LODs, are the candidate sets. The *Cost* and *Benefit* functions define the size and profit of each element. The goal is to select the object tuples with the highest cumulative benefit, while keeping the cumulative potential rendering cost under the target frame time.

Funkhouser and Séquin first implemented a variation of this Algorithm 1 with a runtime of $O(n \log n)$ for n visible objects and a solution that is at least half as good as the optimal outcome.

In contrast, the advanced Algorithm 2 of Funkhouser and Séquin uses information from previously calculated frames due to frequent redundancies in succeeding frames. LODs from the previously calculated frame are reused as the starting point for the algorithm. Thus, only the LODs of a few objects need to be increased or decreased to match the maximum cost calculation. The algorithm retains its runtime complexity of $O(n \log n)$. However, its average runtime is significantly lower than the runtime of the authors’ simple, greedy implementation 1. At the beginning of the program, all nodes are initialised with their minimum LOD. For each frame the LODs of the objects with the highest subsequent *Values* 2.6 are iteratively increased until the cost limit is exceeded, if the LOD is not on maximum. While the calculated rendering costs of the scene are greater than the cost limit, the LODs of the objects with the lowest current *Value* are decreased by one, if the LOD is not on minimum. The algorithm terminates, when the same object is increased and decreased

2. Fundamentals and Related Work

in the same iteration.

$$Value(O, L, R) = Benefit(O, L, R) / Cost(O, L, R) \quad (2.6)$$

Algorithm 1 Primitive implementation of Funkhouser and Séquin’s algorithm.

```
1: For each frame:
2: sort object tuples by value(O,L,R) in descending order into list L
3: while (rendering cost of scene < cost limit) do
4:   Add next object tuple from L to S
5:   if object already in S then
6:     delete the one with smaller benefit
7:   end if
8: end while
```

Algorithm 2 Funkhouser and Séquin’s advanced algorithm interpreted by Nirenstein and Winberg [NW98].

```
1: Initialize at begin of program:
2: for all nodes in scene do
3:   set to min LOD
4: end for
5: For each frame:
6: repeat
7:   Find object M with highest subsequent value && M not max LOD
8:   if M exists then
9:     Increment LOD of M
10:  end if
11:  while (rendering cost of scene > cost limit) do
12:    Find object L with lowest current value && L not min LOD
13:    if L exists then
14:      Decrement LOD of L
15:    else
16:      Exit
17:    end if
18:  end while
19: until M==L
```

2.2.2. Work based on Funkhouser and Séquin

In this subsection I present two papers which improve upon the work of Funkhouser and Séquin.

Gobbetti and Bouvier’s algorithm

The core of the approach by Gobbetti and Bouvier [GB99] is also a cost-benefit calculation, as proposed by Funkhouser and Séquin. Gobbetti and Bouvier provide some enhancements

in the cost calculation. Those adjustments include the different steps of calculation adding up to the rendering time. Table 2.2 gives an overview over the abbreviations used by Gobbetti and Bouvier.

Abbr.	meaning
S	set of objects visible in scene
r	resolution (LOD at Funkhouser and Séquin)
W	viewing configuration (rendering setting at Funkhouser and Séquin)
M	mesh (object at Funkhouser and Séquin)
$t^{(desired)}$	framerate which should be achieved
T	time needed for calculation in various steps

Table 2.2.: Abbreviations used by Gobbetti and Bouvier

In their algorithm they maximize

$$Benefit(W, S(r))$$

with subject to

$$\begin{aligned} Cost(W, S(r)) &\leq t^{(desired)} \\ r &\succeq 0 \\ r &\preceq 1 \end{aligned} .$$

The relational operators \succeq and \preceq denote componentwise inequality and 0 and 1 are constant vectors. The variable $t^{(desired)}$ is the target display time, r is the resolution, that is the LOD at Funkhouser and Séquin, and W is the viewing configuration with lights and camera. The cost of a mesh M with resolution r is then computed similarly to formula 2.3 of Funkhouser and Séquin with

$$Cost(M, W, r) = T^{(setup)} + \max\left(T^{(tri)} \cdot r \cdot \maxtri(M), T^{(pix)} \cdot Coverage(M, W)\right). \quad (2.7)$$

$T^{(setup)}$ is added in respect to the time needed for setting up the rendering environment per object. An example in OpenGL would be the binding of the material. $T^{(triangle)}$ is the time a single triangle of a mesh needs to traverse the rendering pipeline. $T^{(pix)}$ represents the time to calculate the value of a pixel and $Coverage(M, W)$ is the estimated area the whole mesh M covers on the screen under viewing configuration W . Thus, the minimal resolution $r^{(min)}$ can be derived from formula 2.7. This describes the resolution under which no further performance gain can be achieved:

$$r^{(min)} = \frac{T^{(pix)} \cdot Coverage(M, W)}{T^{(tri)} \cdot \maxtri(M)}. \quad (2.8)$$

The cost estimation is then written:

$$\begin{aligned} Cost(W, S(r)) &= T^{(fixed)} + t^{(max)\top} \cdot r \\ r &\succeq r^{(min)} \\ r &\preceq 1 \end{aligned}$$

2. Fundamentals and Related Work

with

$$T^{(fixed)} = T^{(init)} + T^{(final)} + \sum_i T_i^{(setup)}$$

being the part of the frame time. $T^{(fixed)}$ does not depend on the resolution. $T^{(init)}$ and $T^{(final)}$ stand for the initialising and finalising time in the frame. Here, $t^{(max)}$ is the vector, which contains the maximum rendering time $T^{(tri)maxtri}(M)$ for each mesh M and $r^{(min)}$ is the result of equation 2.8. $T_i^{(setup)}$ stands for the time each object i requires for setup. The constants $T^{(setup)}$, $T^{(tri)}$, $T^{(pix)}$, $T^{(init)}$ and $T^{(final)}$ need to be determined by benchmarking in a preprocessing step analogue to the determination of the coefficients C_1 , C_2 and C_3 from Funkhouser and Séquin in equation 2.3. To avoid unwanted changes in frame rates due to the influence of other sources outside the rendering process (other running programs, background processes) a worst case estimate is added to $T^{(fixed)}$.

The authors modified the benefit equation by Funkhouser and Séquin to:

$$Benefit(W, S(r)) = \sum_i Importance(S_i, W) \cdot Accuracy(S_i, r_i)$$

with

$$Importance(M, r) = Coverage(M, W) \cdot Focus(M, W) \cdot Semantics(M)$$

and

$$Accuracy(M, r) = \sqrt{N_v^{(max)} r}.$$

S is the set of visible objects under resolution r with viewing configuration W and M is the mesh. $Importance(M, r)$, in contrast to its definition in Funkhouser and Séquin’s algorithm 2.4, consists of $Coverage(M, W)$, $Focus(M, W)$ and $Semantics(M)$. $Coverage(M, W)$ is the area in pixel covered by the object, $Focus(M, W)$ stands for the distance of the object to the center of the screen and $Semantics(M)$ is a user definable factor for de- or increasing the importance. $Semantics(M)$ is the same as $Importance$ at Funkhouser and Séquin. $Accuracy(M, r)$ is determined by the authors to get good visual results for $N_v^{(max)}$ vertices in a mesh at its highest resolution. $Hysteresis(M, r)$ can be optionally added, but is not included in the evaluation in this paper.

Unlike the algorithm by Funkhouser and Séquin the authors use a resolution $r^{min} + \epsilon$ for a small ϵ as a starting point. Additionally, for frames > 0 , the resolution of the previous frame can be used as a starting point of a frame. The resolution is iteratively reduced by a factor α , beginning at the objects with the lowest cost/benefit ratio. The program terminates, when the constraint is met or when all objects are set to their lowest LOD.

Howell et al.’s algorithm

Howell et al. [HCSS99] propose an algorithm, which they call “market model concept”. The underlying idea is to not calculate the resolutions of each object externally, but let each object calculate its need itself. The value is positive, if it needs more time than in the previous frame and negative, if it needs less time. These variations can occur if the object’s distance to the camera increases or decreases. The algorithm itself chooses pairs which can trade their time. Usually, objects which are no longer visible sell their time to those who are in need. The algorithm finishes either if an object would trade with itself or if no pair can be found. The former case can occur, because an object with a higher need can trade

also with an object with a lower need, if all non-visible objects have already sold their time. The respective market need for the pair of objects is updated after each iteration of trading, as is the LOD. The LOD is set to the highest possible value which can be processed within the new traded time. The algorithm was implemented to run on a single processor machine and is therefore serial. The authors state, that on a multiprocessor machine, the algorithm can be parallelised.

Howell et al. found that their own algorithm had more inaccuracies in the cost implementation, as it showed a number of 116 overrun frames, while Funkhouser and Séquin only had 41 overrun frames. Figure 2.7 shows a comparison of Howell et al.'s implementation of Funkhouser and Séquin's algorithm, their own and two other implementations. The market model concept reveals to have a better performance than the other two implemented algorithms, but it is not significant which one of Funkhouser and Séquin and Howell et al. is better.

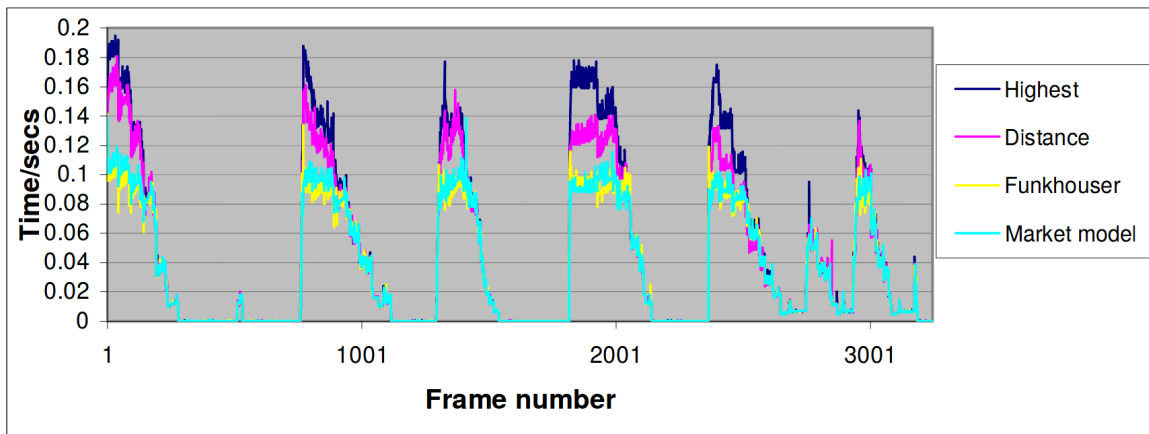


Figure 2.7.: Rendering times for the market model concept, Funkhouser and Séquin's algorithm, a rendering on highest LOD and an algorithm determining LODs via a distance algorithm, implemented by Howell et al. [HCSS99].

3. Concept

In this chapter, I describe how the experiments of my work are designed and which aspects I have in mind.

3.1. Previous considerations

In the paper of Funkhouser and Séquin [NW98] the *Benefit* and *Cost* formulas use $Pix(O)$ respectively $Size(O)$. The later one is further described as an estimate of the number of pixels covered by the object.

At first glance, they might seem the same. With the rendering pipeline in mind, it could mean a practical difference. It would be adequate to use an estimation of pixels for the $Size(O)$ in *Benefit* to determine the significance of an object for a scene at a certain view. But for the *Cost* it would be advantageous to use fragments as part of an estimation of rendering time. A given object could have more rendered fragments than actually shown pixels. Depending on the given layers of an object, which also could be translucent, this would lead to an increasing rendering time. Therefore, I present a concept for an estimation of both fragments and pixels of an object.

3.2. Integration of Funkhouser and Séquin's algorithm

The idea is to implement Funkhouser and Séquin's algorithm in a way, that it can be activated or deactivated, depending on the user's needs. Funkhouser's algorithm uses information from the previous frame for LOD calculation. The dependence is shown in Figure 3.1. Information, such as the last LOD, about the last frame is saved in the respective object nodes, such as last LOD. Figure 3.2 gives an overview, how the algorithm is embedded into the whole system. The algorithm of Funkhouser and Séquin

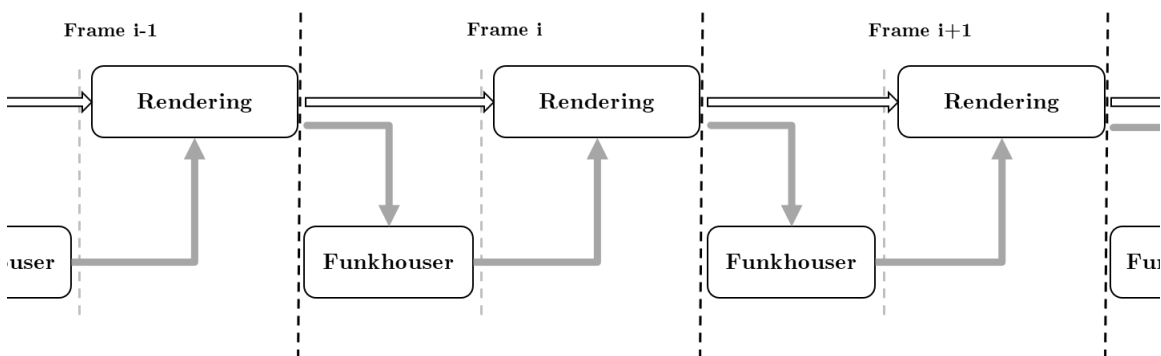


Figure 3.1.: Chart of the integration of Funkhouser and Séquin's algorithm.

3. Concept

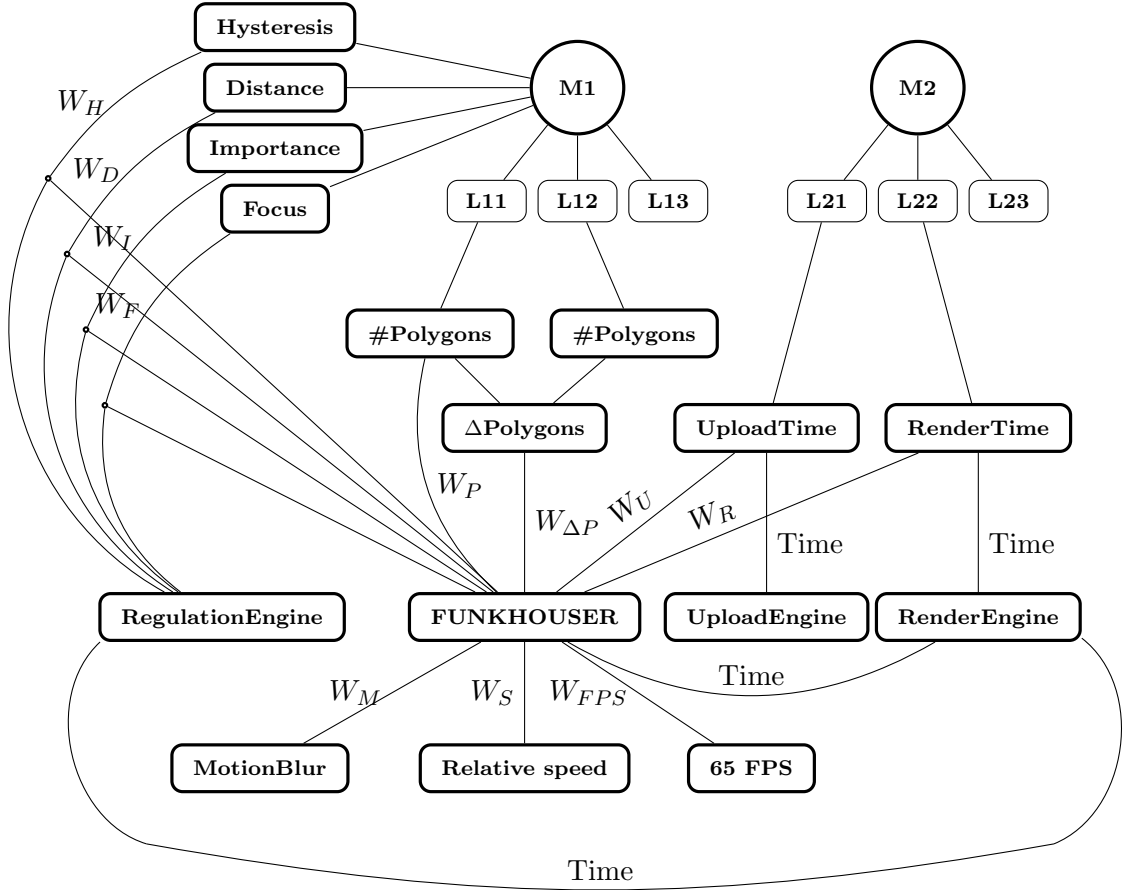


Figure 3.2.: Diagram of the implementation of Funkhouser and Séquin’s algorithm into the existing framework *screenLib*.

3.3. Adaption of Funkhouser and Séquin’s cost formula

The cost formula (see equation 2.3) by Funkhouser and Séquin summarizes the polygon count and the vertex count (multiplied by their respective coefficients). The sum is compared to the pixel count. While determining the coefficients, the polygon count as well as the vertex count is plotted on its own against the time used to render. Therefore, it is the idea to change the formula to:

$$Cost_{PVP}(O, L, R) = \max\left(C_1 \cdot Poly(O, L), C_2 \cdot Vert(O, L), C_3 \cdot Pix(O)\right)$$

The costs calculated by Funkhouser and Séquin would be overestimated all the time, leading to an underachieving selection of LODs. With this adjustment, more high resolution LODs should be selected on runtime.

3.4. Estimation of fragments

Depending on the rotation and size of an object and its distance to the camera, the actual count of fragments can vary. Also, the fragments for the current model-view-projection matrix of an object need to be at hand before Funkhouser's algorithm takes place and, therefore, before the actual rendering process starts. On a practical basis, the fragments themselves must first be processed through the rendering pipeline before they can be counted.

In order to get a most accurate estimation, a lookup table is created in a preprocessing step. For this, a few requirements are needed. First, the object needs to be rendered as big as possible on the screen. Second, not every possible rotation is realistic to be preprocessed for every object in a scene. Hence, I choose a step size of 10° for the rotations of each axis and also to reduce the preprocessing time. Third, rotations on two Euler axis are enough. If only rotations around the X- and Y-axis are chosen, any additional rotation around the Z-axis adds nothing to the information gained. Any plane facing the screen would only be rotated around its centre, as Figure 3.3 shows. The second and third point include both a great reduction of memory space and processing time. Table 3.1 shows an example of the structure of a lookup table for two rotation axes. To project the object as big as possible on the screen, the camera and the object itself need to be positioned accordingly. The exact method is described in subsection 4.3.2.



Figure 3.3.: Visualisation of rotation on Z-axis. Approximately, no changes of amount of pixels are recognised.

X-rot \ Y-rot	Y-rot				
	0°	10°	20°	30°	...
0°	306	304	303	294	...
10°	310	307	306	297	...
20°	307	304	304	295	...
30°	298	295	294	290	...
...

Table 3.1.: Excerpt of a lookup table with placeholder fragment counts per angle.

3.5. Estimation of pixels

I present two options of estimating pixels, whose characteristics I explain in the following section.

3.5.1. Lookup table

For estimating pixels, the same challenge arose as for estimating fragments. To get the exact amount of pixels of an object, the pixels need to be rendered first. But for processing the rendering with Funkhouser and Séquin's algorithm, this information must already be present. Therefore, I am using a lookup table with the same requirements as lookup table for the fragments. To get the camera and the object to a position, where the provided area of the window is used optimally, further computation needs to be done. The exact way of camera and object positioning is described in subsection 4.3.2.

3.5.2. Bounding box

Another possibility to summarise the number of pixels of an object is to estimate them via the size of its bounding box. Hereby, the challenges are to determine the screen coordinates from a given point in the scene. The most 3D applications use the perspective projection, because this is the optimal projection for the human eye, because the distant objects appear smaller [Bla90]. However, the optimal projection for transforming a point given in three dimensional coordinates to two dimensional coordinates is the orthogonal perspective. Figure 3.4 shows how much an estimation via bounding box can differ from the actual pixels of a sphere. Depending on the rotation of the object and, therefore, of the bounding box, it can differ even more. This is only an example for the variance in orthographic projection. Figure 3.5 shows how this behaves for perspective projection. The greatest difference between the orthographic and the perspective projection is in the one-sided view. The three-sided view is at a similar level of overestimating the area calculation via bounding box for both projection forms. Figure 5.1 shows an exact evaluation of the pixels estimated via bounding box and the actual amount covered by the object.

The estimation via bounding box is, therefore, expected to be much more inaccurate than the method via lookup table. However, its advantage is, that it does not need any preprocessing or pre-rendering step and can be calculated within Funkhouser and Séquin's algorithm at run time. For computing the amount of pixel covered by the bounding box, I first get the 2D window coordinates from every corner of the bounding box and then calculate the respective convex hull of these points, as induced by Zachmann [Zac16, p. 21]. With the resulting points, it is simple to determine the covered area by dividing the polygon into triangles and calculate and sum up their respective areas.

3.6. Visualisation

I use different colours to visualise the differences between the LODs. This section deals with the determination of the colour scheme and the mapping of colours to the respective level of details.

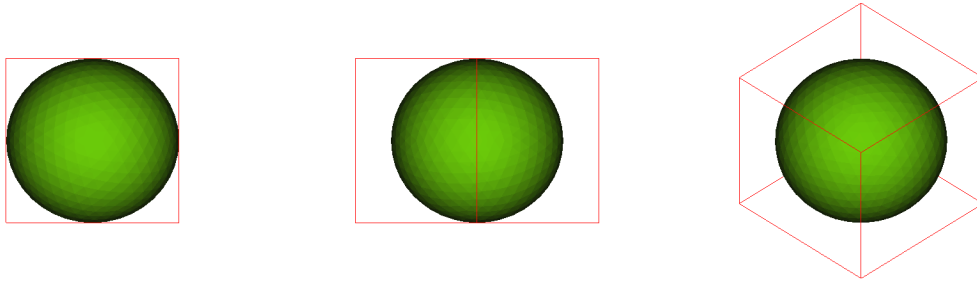


Figure 3.4.: Orthographic projection of an icosphere with bounding box in red from three different angles. from left to right: front view, 90° rotation around Y-axis, 90° rotation around Y-axis and by 90° rotation around X-axis.

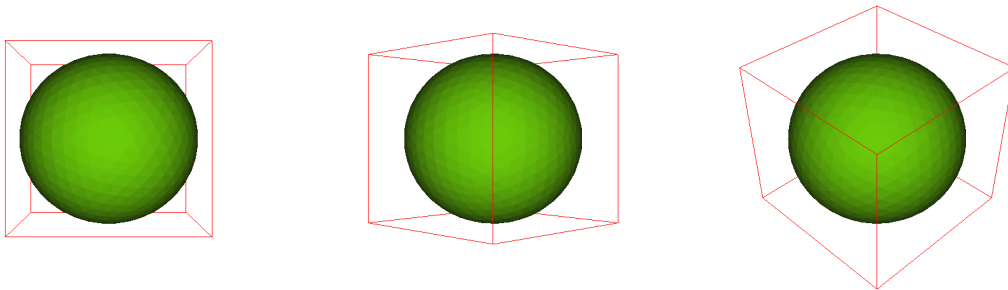


Figure 3.5.: Perspective projection of an icosphere with bounding box in red from three different angles; from left to right: front view; 90° rotation around Y-axis; 90° rotation around Y-axis, followed by 90° rotation around X-axis

3.6.1. Colour scheme

I use the web application *ColorBrewer 2.0* [BH13] for the visualisation of the different LODs in the test scene. This tool reimplements *ColorBrewer.org* by Harrower and Brewer [HB03]. The program creates colour tables for ranges from 3 to 12 data classes under a number of requirements. For the LODs, the following requirements are given: 8 *sequential* data should be illustrated as *colourblind safe*, *print friendly* and *photocopy safe*.

When more data classes are chosen, the selected colours can become less distinguishable. Thus, the web application does not show any results for the requirements above. Therefore, it is necessary to scale down or alter the demands. First, to have a clear distinction between the lowest and the highest level of detail, the selection *sequential* data is changed to *diverging*. Second, *photocopy safe* and *print friendly* are deactivated and only *colorblind safe* is kept. From the resulting colour schemes the one with colours ranging from red to blue, respective warm to cold, as shown in Figure 3.6, is chosen.

3. Concept

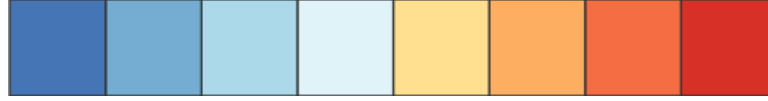


Figure 3.6.: The colour scheme used for visualisation of LODs. Dark blue mapped to the lowest LOD, dark red mapped to the highest LOD. The respective color values in RGB from left to right: [69,117,180], [116,173,209], [171,217,233], [224,243,248], [254,224,144], [253,174,97], [244,109,67], [215,48,39].

3.6.2. Colour mapping

For 8 LODs per object, the lowest LOD is mapped to the darkest blue and the highest LOD to the darkest red. The LODs ranging between are mapped with the respective lighter colour shades. As the 8 colours are to be implemented as constants and for flexibility reasons, a mapping function for less than 8 LODs is needed.

$$colour(i) = \begin{cases} i \cdot \frac{colour_{max}}{LOD_{max}}, & \text{if } i < \frac{LOD_{max}}{2} \\ colour_{max} - (LOD_{max} - i) \cdot \frac{colour_{max}}{LOD_{max}}, & \text{if } i \geq \frac{LOD_{max}}{2} \end{cases}$$

In this equation, $colour_{max}$ stands for the last position of the array of colours and LOD_{max} is the last position of the amount of LODs. The variable i is the number of the LOD to be mapped to its respective position. This gives a function where all input values, respectively LODs, are evenly spread with a maximum distance over the available colours. Should there be only one LOD provided, or should there be more LODs than available colours, the vertex colour changing function is not called.

4. Implementation

This chapter is about the implementation of the system. In the following, I explain challenges, which arose during the implementation of the presented algorithms, and their solutions. I also present frameworks and libraries which are used within the implementation.

4.1. Included libraries and environment

I enhanced the *screenLib*, a library created at the *Leibniz Supercomputing Centre (LRZ)*. It implements a rudimental scenegraph with nodes. Those nodes can get multiple LODs assigned.

The visualisation is based on *OpenGL* and its shader language *GLSL*. The import of 3D models is outsourced to the *Open Asset Import Library (Assimp)* [Tea17]. This is a framework especially for handling 3D models of a various amount of data formats. The export of several single 3D files describing multiple LODs (and timesteps) of the same single object into a M3D-file is done by *Assimp*, as well. With this intermediate step, future running processes of the respective program can load those faster into the memory of the workstation.

4.2. Implementation of Funkhouser’s algorithm

This section is about the implementaton of Funkhouser and Séquin’s [FS93] advanced algorithm and its integration into *screenLib*.

4.2.1. Determining coefficients

As described in the previous chapters, the core of Funkhouser’s algorithm is a benefit-cost-calculation:

$$Cost(O, L, R) = \max\left(C_1 \cdot Poly(O, L) + C_2 \cdot Vert(O, L), C_3 \cdot Pix(O)\right)$$

Some coefficients are needed to be calculated before computing the costs. Each needs to be done individually for every machine on which the algorithm will run on. The program is tested with two shader scenarios. The first is a very basic Phong shader. The second is a the same shader but with an artificially expanded part in the fragment shader. This is done, to provoke the rendering time to be depended also on the number of fragments and pixels. In the future, it can be easily adjusted to other shaders. When rendering many different LODs of the same object, the according rendering time needs to be measured. Figure 4.1 shows the rendering time as a function of the polygon number for a certain machine (*Intel Xeon(R) CPU E5-1620 v3 @ 3.500 00 GHz × 8, GeForce GTX 980 Ti/PCIe/SSE2*). Figure 4.2 shows the rendering time as a function of the vertex number for the same machine. A function for the best fitting line through all given data points had to be determined and the corresponding

4. Implementation

coefficients were calculated. Table 4.1 gives an overview of all coefficients and their values. Figures 4.3 and 4.5 show the corresponding coefficient determination under the basic shader for fragments and pixels, respectively.

Using a simple rendering algorithm, the diagrams 4.3 and 4.5 show almost no dependency on fragments and pixels. But a higher correlation between the chosen LOD, and consequently the number of polygons, and its corresponding rendering time is given. The data points build rather parallel lines to the y-axis. With the expanded rendering algorithm, the time shows also dependency on fragments 4.4 and pixels 4.6, the gradients show still a dependency on the LODs. This leads me to the conclusion, that the original formula of Funkhouser and Séquin has the potential to be improved:

$$Cost_{PVP}(O, L, R) = \max\left(C_1 \cdot Poly(O, L), C_2 \cdot Vert(O, L), C_3 \cdot Pix(O)\right)$$

Rendering	Coefficient	Value
Phong	C_1	0.000 993 95
Phong	C_2	0.001 987 90
Phong	C_{3frag}	-0.000 001 50
Phong	C_{3pix}	0.001 262 20
expanded FS	C_1	0.000 994 17
expanded FS	C_2	0.001 988 30
expanded FS	C_{3frag}	0.064 823 00
expanded FS	C_{3pix}	0.149 830 00

Table 4.1.: Coefficients determined via rendering of exemplary objects for regular Phong shading and the expanded fragment shader.

4.2.2. Funkhouser’s advanced algorithm

The advanced algorithm developed by Funkhouser needs a few container classes for sorting the node objects accordingly (see Figure 4.7). The first multimap encloses the *subsequent values* of the node object in decreasing order as keys. If those computed values are the same, the object’s address is used to create a total order on all elements in the map. The value of the multimap is then a pointer to the object. The second multimap contains *current value* in increasing order as its key. Analogue to the first multimap, the address of the object is used to create a total order. The value of the respective map-items is also a pointer to the original object. An unordered map combines those multimaps. The unordered map consists of a node object as key and a pair of pointers to multimaps as value.

This construct is in use, to be able to update and sort the nodes accordingly to the *current value* and the *subsequent value*. It also grants access from the last sorted value in one list to the respective object in the other list. The *value* is the result from the *Benefit/Cost* calculation.

4.2. Implementation of Funkhouser's algorithm

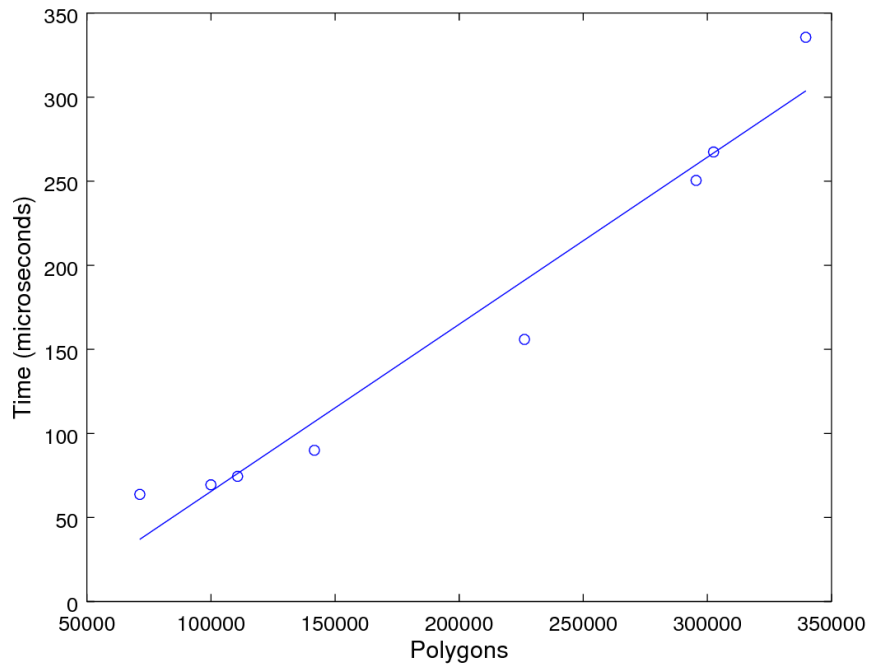


Figure 4.1.: Rendering time relative to the amount of polygons.

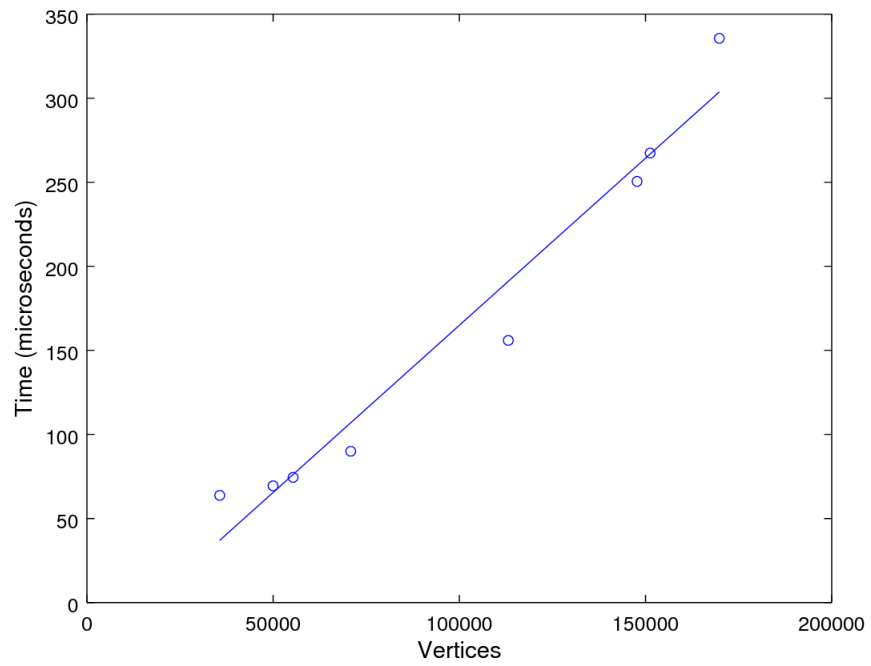


Figure 4.2.: Rendering time relative to the amount of vertices.

4. Implementation

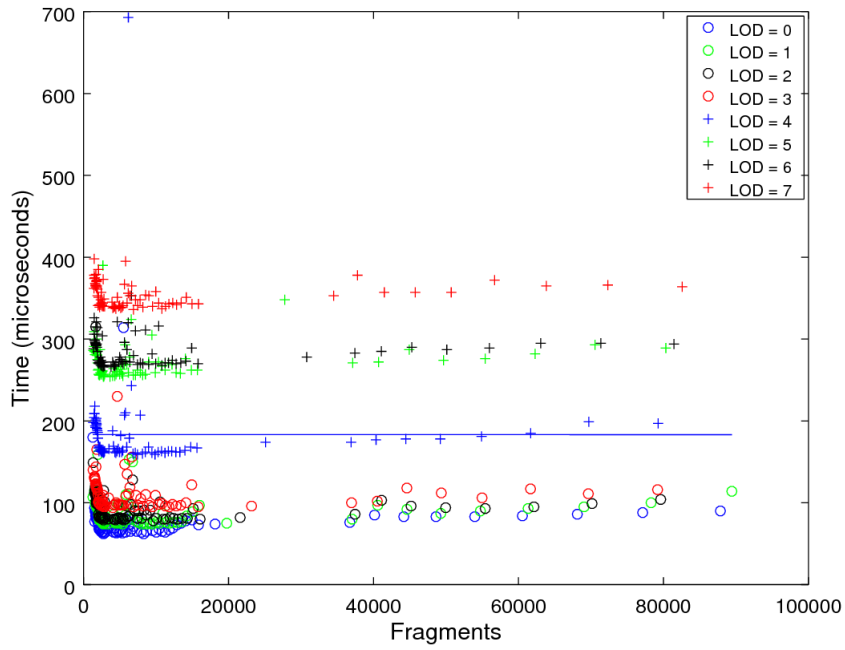


Figure 4.3.: Rendering time per amount of fragments.

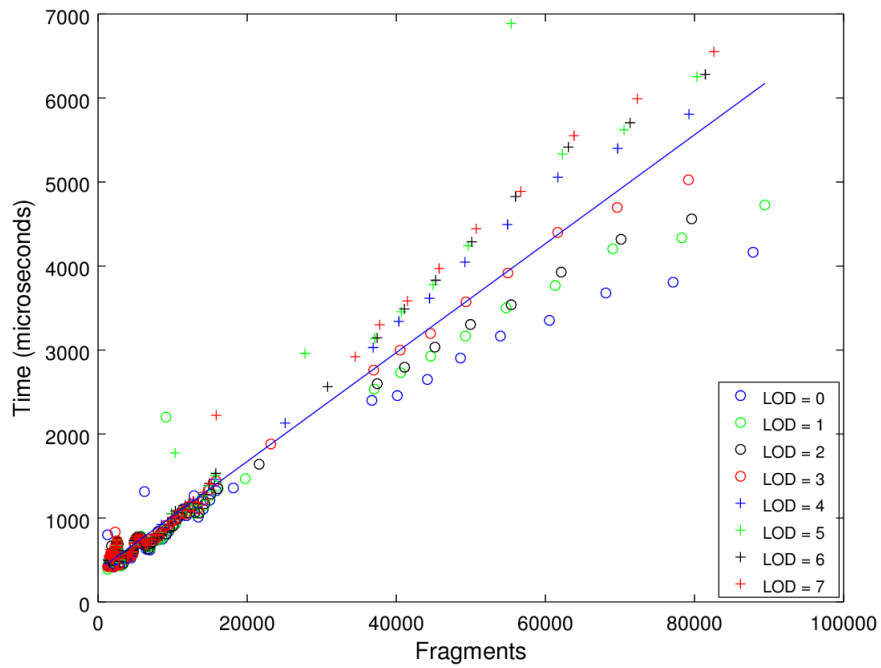


Figure 4.4.: Rendering time per amount of fragments with expanded fragment shader.

4.2. Implementation of Funkhouser's algorithm

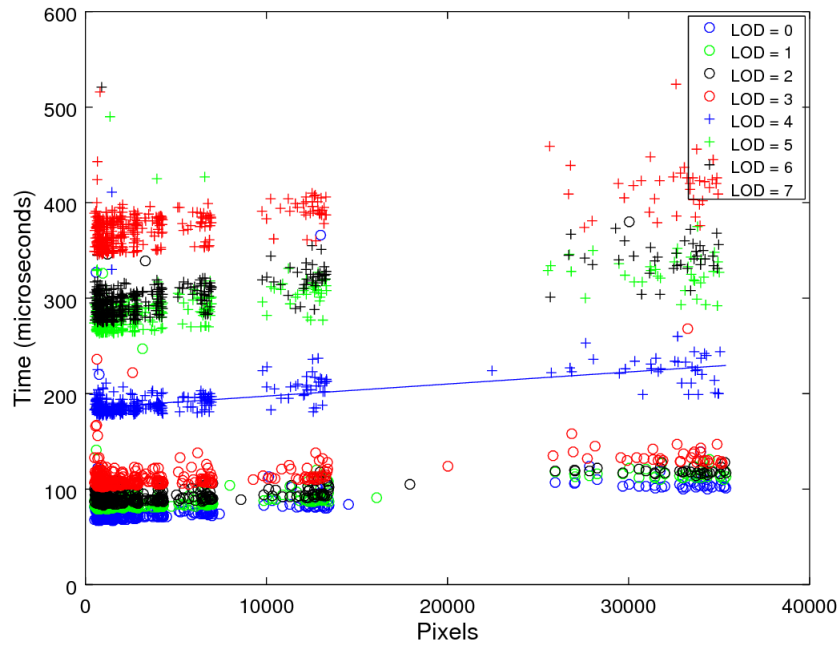


Figure 4.5.: Rendering time per amount of pixels.

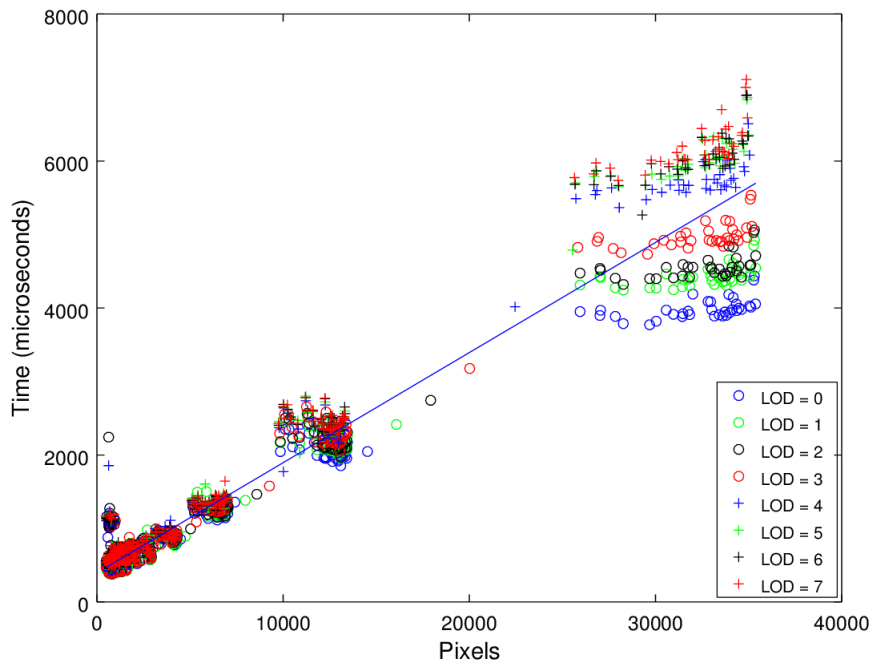


Figure 4.6.: Rendering time per amount of pixels with expanded fragment shader.

4. Implementation

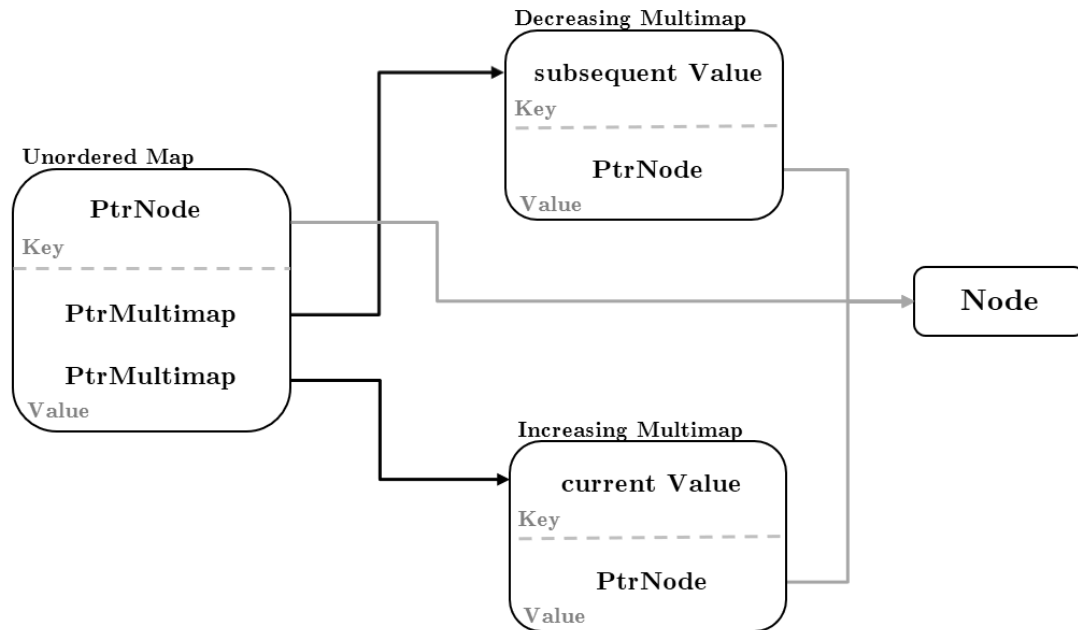


Figure 4.7.: Diagram with the construct of one unordered map and two multimaps. The connections via pointer allow to maintain access from one specific object within one map to the same object within the other.

4.3. Size estimation features

This section is about the size estimation features I implemented for the rendered objects. They are needed to calculate the results of *Cost* and *Benefit*, respectively. As mentioned in chapter 3, there are three different methods to determine the size of an object: First, there is pixel estimation via bounding box, second, there is pixel estimation with a lookup table and the third one is fragment estimation with a lookup table.

4.3.1. Estimation via bounding box

The estimation of the size of an object via bounding box is relatively cost-efficient, because the bounding box of an object can be created very fast and its memory requirements are not very high. The algorithm for calculating the screen area takes the eight corner points of the bounding box. It projects them from 3D space world coordinates into 2D space world coordinates. The results are the respective window coordinates in pixel. Then the program calculates the convex hull of this point set and determines the covered area.

There is a maximum of only 8 points for each object for which the convex hull is calculated. Therefore and for implementation reasons I am using the gift wrapping algorithm by Jarvis [Jar73]. It has $O(nh)$ time complexity, where n is the number over all points and h is the amount of points on the convex hull. In my implementation, this leads to a worst case of $O(8 \cdot 6) = O(48)$. $O(48)$ is a constant runtime complexity and does not influence the overall time complexity of my implementation of Funkhouser's algorithm. It still depends only on

the amount of objects and their respective LODs in the scene to be rendered.

After the convex hull is computed, it is necessary to calculate the points which are within the window size. For this purpose, I use the polygon clipping algorithm of Sutherland and Hodgman [SH74]. The general runtime complexity of this algorithm is $O(ev)$, where e is the number of edges of the clipping plane and v is the number of vertices of the polygon to clip. In my case, the clipping plane is always the window with four edges. The polygon to clip, the former convex hull, has got six vertices in worst case and four vertices in best case. This leads to a worst case of $O(4 \cdot 6) = O(24)$. This is a constant, too, and does not influence the runtime complexity of Funkhouser and Séquin's algorithm.

The algorithm of Sutherland and Hodgman determines the orientation of three points. It calculates, if they are oriented co-linear, clockwise or counter-clockwise. This depends not on the value, but rather on the respective sign. I use the crossproduct to determine this. When calculating the crossproduct, the algorithm uses multiplication, which can happen between two very small or two very large values. To prevent overflow as well as underflow, each coordinate of the three 2D-vectors is normalised by dividing it by the maximum absolute value among the corresponding six coordinates.

4.3.2. Preprocessing and lookup tables

To get a more precise estimation of the area which an object covers, I determine the area of pixels and fragments. As this takes additional processes of rendering, a preprocessing step creates lookup tables. The preprocessing steps for creating the lookup tables of fragments and pixels are set up very similarly to each other. For each object used in the scene and for each individual LOD, the object is rendered while rotated a certain step width along two Euler axes. After each rotation step the respective units were counted and written into the lookup table.

Pixel Lookup Table

The pixel lookup table is created by rendering the whole object from different perspectives. It is completely rendered in black to distinguish it from the grey background color. The algorithm calls a function for the whole window, and for each pixel registered as black, a variable is increased. The resulting value is written into the file of the lookup table with the corresponding rotational coordinates of the object. This way, the program can find this value later within the Funkhouser algorithm.

Fragment lookup table

The finding of the amount of fragments covered by an object needs another function. I use an atomic variable in the fragment shader, which is incremented for each fragment, that is each call of the fragment shader. Within the display function, there is an corresponding call, which reads the atomic counter into a local array. After this, the program writes the result into the corresponding file containing the lookup table with the corresponding rotational coordinates of the object for finding the value later within the Funkhouser algorithm.

4. Implementation

Object positioning

To get a good view on the object and, therefore, a good approximation of the amounts of pixels and fragments for the respective lookup table, I place the object in the center of the world coordinate-system. The algorithm translates the object back to the centre of the world via accessing its bounding box coordinates. The X-, Y- and Z-translation is computed separately by taking the minimum and the maximum value of each axis. Those formulas are:

$$\begin{aligned}x_{trans} &= \frac{x_{max} - x_{min}}{2} - x_{min}, \\y_{trans} &= \frac{y_{max} - y_{min}}{2} - y_{min}, \\z_{trans} &= \frac{z_{max} - z_{min}}{2} - z_{min}.\end{aligned}$$

Camera positioning

My goal is, not only to have the object centred to the screen, but also to get a complete view of it for determining the pixels and fragments. Therefore a good estimation of those pixels and fragments is possible with nearly every translation and rotation of the object within the run of the actual program.

There are two possibilities to get the object best suitable into the window and to maximize the computed fragments or pixels: either to scale the object to a computed size with a given distance for the camera given or to compute only the distance of the camera. As the first one needs the a distance to be determined, too, I implemented the latter one. The opening angle of the camera is set to $2\phi = 45^\circ$ by default and the diameter of the surrounding sphere of the object in Figure 4.8 is given by the diameter of the object's bounding box. (In most cases, this is an overestimation of the sphere, but it makes sure, to include the object completely. An alternative would be to compute an individual bounding sphere.) The camera's X- and Y- coordinates are set to 0 to get a straight view on the object with its center-coordinates in (0,0,0). The program computes the Z-value z of the camera as the hypotenuse of a right-angled triangle:

$$z = \frac{a}{\sin(\phi)}.$$

Now, the actual window size is only a part of the cone (see Figure 4.8) given by this computation. The solution is to set the new height, respectively width (depending on which is smaller), as the new aspired diameter of the cone. For this computation, I use the intercept theorem to get the factor for the current Z-value, as shown in Figure 4.9: First, the relation between window height, respectively width, and diameter needs to be found, as width and height are only given in pixels and diameter is represented only in world coordinates:

$$(2a)^2 = w^2 + h^2.$$

The algorithm uses the intercept theorem and the value of $2a$ to get z_{final} , which includes z and the the additional distance c of the camera to the object:

$$z_{final} = z + c = z \cdot \frac{2a}{\min(w, h)}.$$

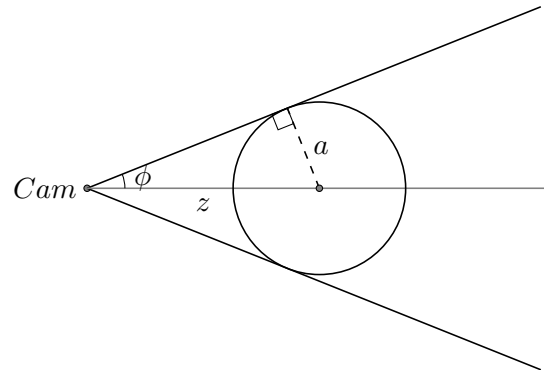


Figure 4.8.: Use of the trigonometric functions for determining the distance z of the camera to the object. $2a$ equals the diameter of the bounding box of an object. The variable ϕ represents the camera opening angle.

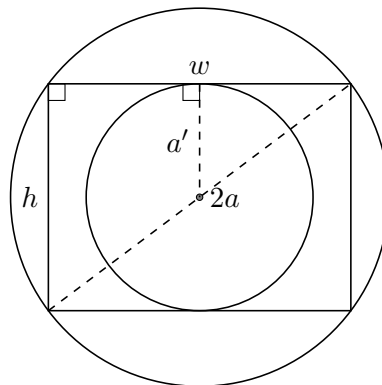


Figure 4.9.: Use of the intercept theorem for exact camera positioning. The variables w and h are equal to width and height in number of pixels. The relation to $2a$ in world coordinates can thus be computed. The variable a' represents the maximum radius of a bounding sphere of an object which still fits in the screen window.

Finding related data in lookup table

The actual program needs to find the related data fragments or pixels within the lookup table. Here, the object's rotation is stored with the object in the centre of the world. Therefore, the current matrix transformation needs to be reversed with the transformation from creating the lookup table to get the current fragments or pixels. This works, because

4. Implementation

of the commutative property of matrix multiplication of the respective inverse:

$$E = A^{-1} \cdot A = A \cdot A^{-1}.$$

Also needed is the associative property respective to matrix multiplication in general:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C.$$

After this, the algorithm does a matrix decomposition. For the search in the lookup table, the respective angles are needed. Therefore, I use an approach proposed by Ho [Ho11], which computes the rotation angles relative to X-, Y-, and Z-axis, meaning Tait-Bryan angles: Let a matrix R have the form of:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}.$$

I compute the angles θ_x , θ_y and θ_z with:

$$\theta_x = \begin{cases} \text{atan2}(r_{32}, r_{33}) + 2\pi & \text{if } \text{atan2}(r_{32}, r_{33}) < 0, \\ \text{atan2}(r_{32}, r_{33}) & \text{otherwise.} \end{cases},$$

$$\theta_y = \begin{cases} -\arcsin(r_{31}) + 2\pi & \text{if } -\arcsin(r_{31}) < 0, \\ -\arcsin(r_{31}) & \text{otherwise.} \end{cases},$$

$$\theta_z = \begin{cases} \arctan(r_{21}/r_{11}) + 2\pi & \text{if } \arctan(r_{21}/r_{11}) < 0, \\ \arctan(r_{21}/r_{11}) & \text{otherwise.} \end{cases}.$$

The definition of $\text{atan2}(y, x)$ is given with:

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{if } x > 0, \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{if } x < 0 \text{ and } y \geq 0, \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{if } x < 0 \text{ and } y < 0, \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0, \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0, \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

The implementation of atan2 in `math.h` (see A.1) returns a value in the range of $[-\pi, \pi]$, according to Linux programmer's manual [ata01]. To get results in the range of $[0, 2\pi]$, 2π is added to negative values. I use linear interpolation to compute the weighted average fragment count for rotational angles between two samples. For example, a sample is taken every ten degrees, as in Table 4.2. The computed real rotation is 12° on X-axis and 29° on Y-axis, the average fragment count is computed with the help of a contingency table such as Table 4.3. This leads to the equation $299\ 028 = 0.08 \cdot 305\ 236 + 0.72 \cdot 298\ 916 + 0.02 \cdot 302\ 088 + 0.18 \cdot 296\ 376$.

x-rot \ y-rot	0°	10°	20°	30°	...
0°	305 536	305 652	302 074	295 992	...
10°	309 184	308 778	305 236	298 916	...
20°	306 280	305 554	302 088	296 376	...
30°	297 046	296 476	293 674	289 158	...
...

Table 4.2.: Excerpt of a an actual lookup table with fragment counts per angle.

x \ y	y_{lower}	y_{upper}
x	0.1	0.9
x_{lower}	0.8	0.72
x_{upper}	0.2	0.18

Table 4.3.: Contingency table for computing quotients of fragment counts.

4.4. Program arguments and control keys

The program itself can be executed in seven different modes. The first, *save*, saves all objects given as PLY- or OBJ-files within a TXT-file into an M3D-file. This guarantees faster access when running the further program modes, which only work with those M3D-files. This M3D-file includes all LODs and timesteps of an object. The second mode *load* creates the lookup tables for fragments and pixels, depending on which additional argument is given. The modes *calcCoeffFrag*, *calcCoeffObj* and *calcCoeffPix* create the input data as CSV-files for the computation of the coefficients of the machine for the algorithm of Funkhouser and Séquin.

The *run*-mode shows the actual scene, while calculating the corresponding LODs with Funkhouser's algorithm. Here, the camera position can be controlled with the arrow keys and tilt of the camera can be modified with the keys “.” and “,”. The user can start a tracking shot, where the camera follows a programmed path with pressing “F1”. Pressing “F2” activates the tracking shot, too, but creates files containing computed data within Funkhouser's algorithm, like pixels and fragments via lookup table or pixels via bounding box.

For comparing the estimated fragments and pixels from the the *run*-mode with the actual fragments/pixels, the user can choose the modes *camMovePixelCount* or *camMoveFragmentCount*. Those count the actual fragments or pixels while running the tracking shot and write them in a CSV-file for each frame. Other programs can then process those files.

5. Results and Discussion

In the following chapter, I explain how the implemented system is tested. I also present the respective results from those tests and discuss them.

5.1. Sphere

For basic tests, I used an icosphere created with blender 2.78c [Fou17]. I chose a sphere, because it is the most compact 3D body which can be created. This means, it has the least amount of volume in relation to its surface. This would also provide a similar, if not the same, number of pixels from every angle of view for the sphere. With this, a better comparability to its bounding box from different angles is achieved. A bounding box would deliver a certain deviation in the number of pixels, it covers, depending on the angle of view.

For calculating the accuracy of the bounding box mechanism, I use the screenshots of an icosphere from Figure 3.4 on page 19 and Figure 3.5 on page 19 and process them with the help of the histogram tool of GIMP 2.10 [gim17]. This determines the accurate number of pixel covered by the sphere and enclosed by the convex hull of the bounding box points by counting the blue, respectively the green, pixels. The detailed results are shown in Table 5.1. The one- and two-sided views of the orthographic view (No. 1 and 2) deliver a much more accurate estimation of the real pixel number. Here, the number of the sphere's pixels is only at 78.8%, respectively 71.6%, of the number of pixels of the bounding box. At the three-sided view both bounding box estimations are of similar accuracy and the sphere is overestimated by even 119.6% at orthographic view and by 116.9% at perspective view.

5.2. Test scene

For more extensive testing, I created a scene consisting of a number of Stanford Armadillos [sta14]. The Stanford Armadillo is provided by the Stanford Computer Graphics Laboratory and represents a non-trivial 3D model. One reason for choosing it is, that it is possible to create different LODs from the base model relatively fast. The other reason is, that it can put the GPU under pressure with enough specimens.

5.2.1. Scene components

I created the respective LODs before the actual test. When the program loads them into its memory, it colours the LODs with the colour scheme described in section 3.6. This results in the models shown in Figure 5.1. The lowest LOD is on the top left, coloured in the deepest blue on the used colour range and the highest LOD is on the bottom right, coloured in the deepest red on the used colour range. The LODs between them are coloured in shades between those extreme colours.

5. Results and Discussion


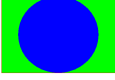
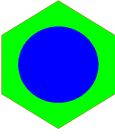
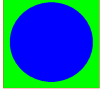
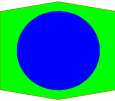
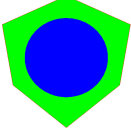
No.	image	bounding box (pixels)	sphere (pixels)	% of bounding box	% overestimated
1		68 864	54 264	78.8	126.9
2		75 746	54 212	71.6	139.7
3		119 032	54 203	45.5	219.6
4		98 532	58 134	59.0	169.4
5		115 300	58 100	50.4	198.4
6		125 982	58 072	46.1	216.9

Table 5.1.: Table of number of pixels of a sphere with and without bounding box. Those are calculated via screenshot and using GIMP 2.8.10 [gim17] and its histogram tool.

5.2.2. Camera simulation

The scene is built with 10×10 armadillos arranged in a square and one additional stray armadillo, as shown in Figure 5.2. This Figure also pictures the route of the camera. The route starts within the separated armadillo and slowly approaches the block of the 100 armadillos. Directly in front of them the camera takes a left turn of 90° . The camera runs along the front row of armadillos, turns right and takes a route directly through the armadillos with another additional slightly right turn at frame 990. After that, the camera is leaving the arrangement and the simulation ends at frame 1220.

Screenshots of the key-frames 0, 290, 380, 480, 590, 990, 1020 and 1220 of the camera

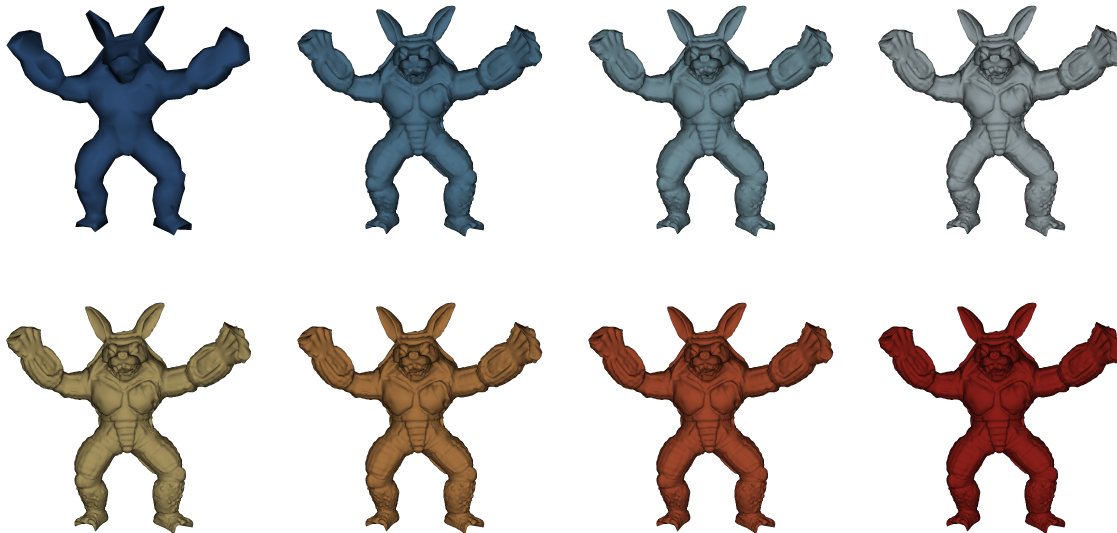


Figure 5.1.: Armadillo in different resolutions. Lowest LOD on the top left, highest LOD on the down right.

simulation path can be found in the Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8. The columns show from left to right the views created by the program with the Funkhouser algorithm with the bounding box, the lookup table via fragments and the lookup table via pixels. The rows show from top to bottom the original cost formula and the PVP formula first with the slim fragment shader, then with the adjusted fragment shader.

5.2.3. Estimation bounding box vs. estimation lookup table vs. real pixels

As pictured in Figure 5.3, the actual pixels over the scene build a graph, which shows a local minimum between frame 400 and frame 500. The same valley is also present within the graph of the pixels estimated via bounding box and within the graph of the pixels estimated via the lookup table created in the preprocessing step. The valey indicates a timeslot within the camera simulation, where the camera first turns 90° to the left and then moves away from the majority of the objects. This leads to a drop in the estimated amount of frames per seconds.

Overall, the bounding box estimation shows a rather similar curve to the actual pixels with both featuring a local maximum between frame 1 and frame 15, then both dropping rapidly and rising again at frame 300. The valley mentioned above follows and rises again until frame 590. Also, most of the followed local maxima and minima occur around the same frames. Those local maxima are at frame 700, frame 750, frame 850, frame 975, frame 1100 and the last one at frame 1175. The respective minima are at: frame 650, frame 760, frame 800, frame 950, frame 1050, frame 1160 and the last one from frame 1190 on.

In contrast, the graph of the pixels estimated via lookup table shows a rather different trend, starting on a slightly higher level (but much lower than the other two graphs) at frame 1 and slowly falling until frame 300. It is then building a local maximum at 325, dropping until frame 500, rising until frame 550, dropping again at frame 560 and rising again with a overall maximum at frame 580. After that, it is nearly constantly falling until frame 1200.

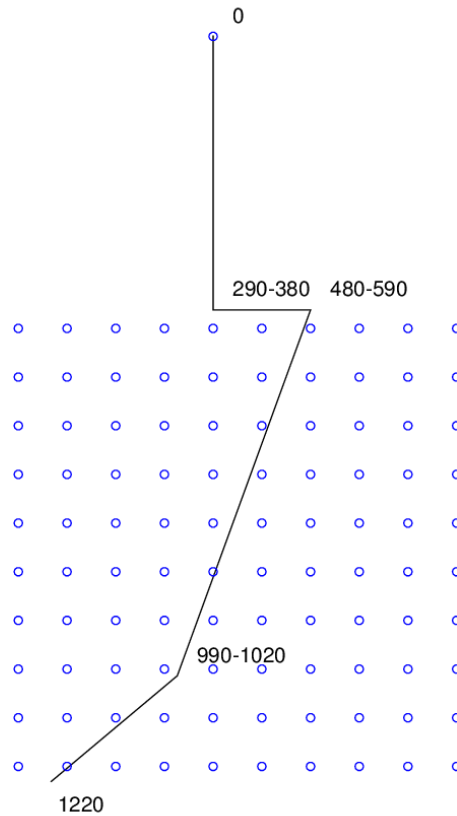


Figure 5.2.: Camera route within armadillo scene with the respective camera positions at frames 0, 290, 480, 990, 1220. The ranges indicate changes in camera direction within these frames.

5.2.4. Estimation bounding box vs. estimation lookup table vs. real fragments

Figure 5.4 shows, that the graph picturing the pixels estimated via bounding box is nearly identical to the graph showing the actual fragments. They not only show maxima and minima at the same frames (see Figure 5.3), but have also the same heights.

Therefore, the maxima for both, the actual fragments and the pixels estimated via bounding box, are at frame 1, frame 300, frame 600, frame 725, frame 775, frame 875, frame 1000, frame 1075 and frame 1175. The minima are at frame 10, frame 500, frame 675, frame 750, frame 800, frame 925, frame 1025, frame 1100 and then from frame 1190 on.

The graph of fragments estimated via lookup table has a very similar shape as the graph of pixels estimated via lookup table. The maxima are at frame 0, frame 320, frame 550 and frame 580. The minima are at frame 300, frame 500, frame 575 and then from frame 1190 on, where the pixel count is 0.

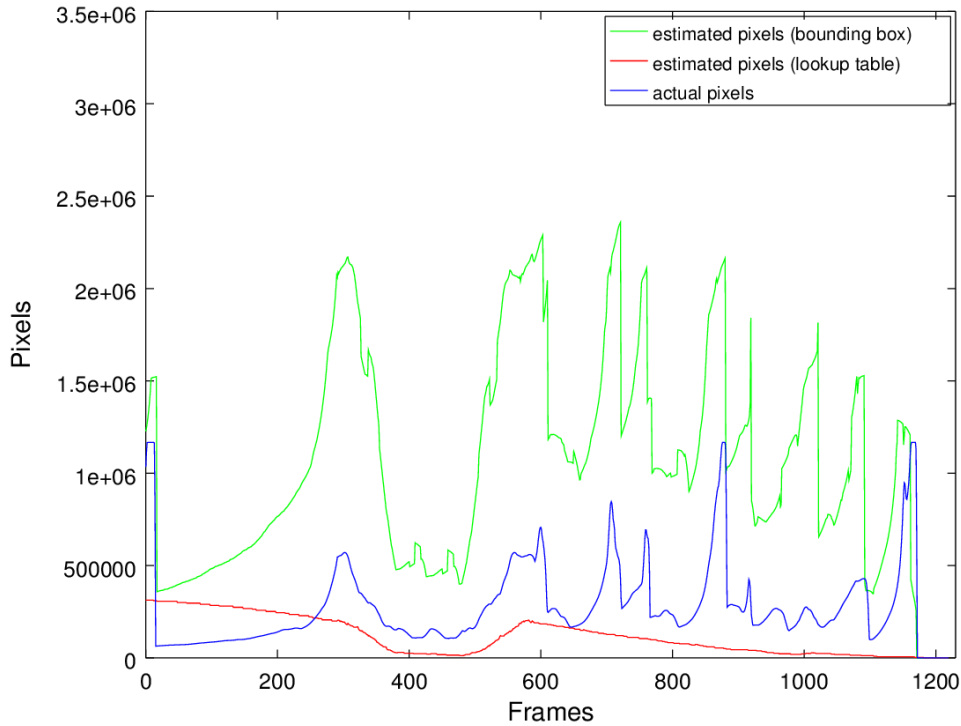


Figure 5.3.: Total amount of pixels per frame. The number of pixels estimated via bounding box are pictured in green, the number of pixels estimated via lookup table are red and the actual pixels are drawn in blue.

5.3. Comparing cost formulas

This section is about the comparison of the two implemented equations:

$$Cost_{original}(O, L, R) = \max\left(C_1 \cdot Poly(O, L) + C_2 \cdot Vert(O, L), C_3 \cdot Pix(O)\right), \quad (5.1)$$

$$Cost_{PVP}(O, L, R) = \max\left(C_1 \cdot Poly(O, L), C_2 \cdot Vert(O, L), C_3 \cdot Pix(O)\right). \quad (5.2)$$

I analysed the equations by running them in different rendering processes. Table 5.2 shows an overview of the exact results, which I gathered while running the tests. None of the two cost equations show a significantly better runtime with the algorithm of Funkhouser and Séquin. The mean frame time with the regular fragment shader is always better the original equation 5.1, whereas the mean frame time is always better using the extended fragment shader in combination with equation 5.2. Within the extended shader renderings, the mean frame time of the versions rendered with the Funkhouser algorithm active are throughout better than with the algorithm inactive and all objects rendered on their highest resolution. The cost estimation of Funkhouser is also responsible for a more flattened estimation of costs. This is observable especially in the top figures of 5.6 and 5.8. Here, the graph of the

5. Results and Discussion

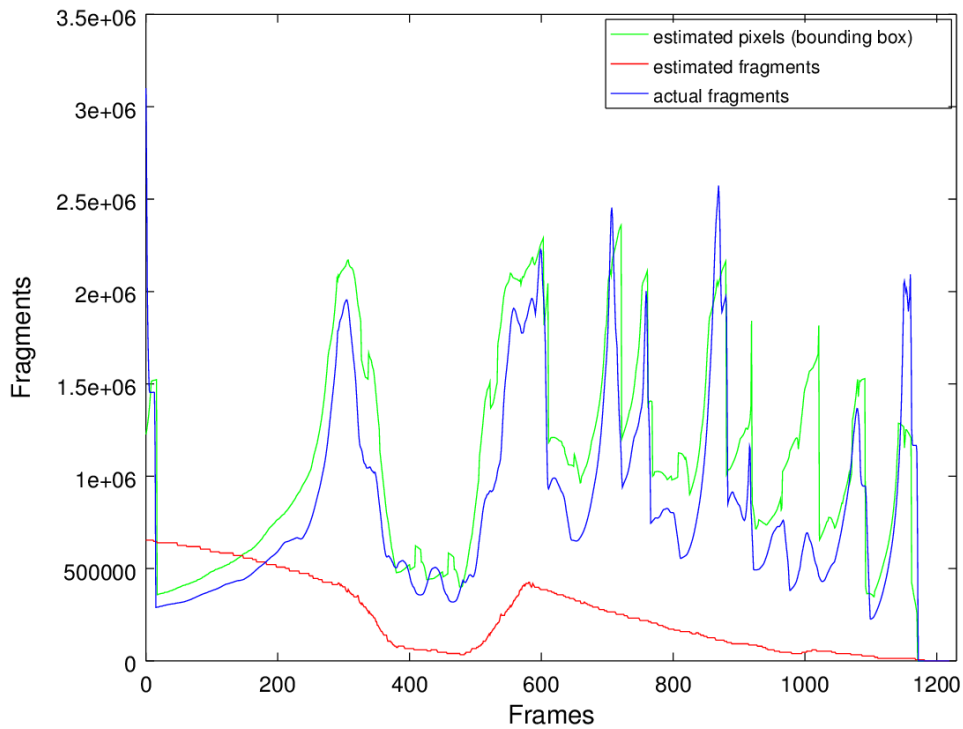


Figure 5.4.: Total amount of fragments per frame. The amount of fragments estimated via lookup table are pictured in red, the amount of the actual fragments in blue and the amount of pixels estimated with bounding box in green.

estimation costs follows the horizontal line of the maximum costs for a bit, whenever it is about to cross it. This leads to several recalculations of LODs, which add computation time.

The Figures 5.6 and 5.8 show, that the calculation of the LODs using the extended fragment shader takes place only at a few frames. That is the case, when the calculated costs cross the line of the maximum costs. All objects in the scene change their LODs to the minimum, respective maximum. None of the objects get LODs between the extrema assigned. The original cost calculation leads to some additional running time of the Funkhouser algorithm between frame 710 and 830 in the top diagrams of Figures 5.6 and 5.8. This adds up to the complete frame time and negatively influences the overall frame time of this run of the program. It is noticeable, that the costs are always estimated to low compared to the actual costs. A rendering on the lowest LOD for all objects crosses almost never the line of the maximum LOD costs and is most of the time at least the double of the maximum costs. In contrast, the rendering with the regular fragment shader shows a much more consistent estimation of the actual frame time, as it is pictured in Figures 5.5, 5.7 and 5.9. This is an indication, that the equation used by Funkhouser and the equation proposed by me is not sufficient for program variations with more extended fragment shaders. Therefore, the formula should be reworked.

Figure 5.10 shows another kind of anomaly. Here, the costs estimated with the bounding box are always calculated too high. They are double to three times higher than the

actual costs. This leads to the fact that there is no recalculation within the algorithm of Funkhouser and Séquin at all. All LODs are already rendered on their respective lowest LOD. The reason for this behaviour is within the computed coefficient for the pixels. The coefficient is determined with separate rendering and counting of the exact pixels. For the cost calculation, there are no further adjustments done, so the same coefficients are used for the cost estimation via the lookup table pixels and the bounding box pixels.

LOD Selection Algorithm	LOD Computing Time (ms)			Frame Time (ms)			
	Min	Mean	Max	Min	Mean	Max	Std. Dev.
None (highest LOD)	0.0	0.0	0.0	0.0	16.479	35.0	8.7201
None (lowest LOD)	0.0	0.0	0.0	0.0	2.3705	7.0	1.1375
orig. BB	0.0	8.4426	137.0	1.0	16.498	152.0	18.431
orig. Pix.	0.0	3.3533	111.0	1.0	10.734	159.0	13.845
orig. Frag.	0.0	3.0631	110.0	1.0	10.514	156.0	13.612
PVP BB	0.0	8.7492	247.0	1.0	19.474	266.0	24.868
PVP Pix.	0.0	2.7123	106.0	1.0	12.933	215.0	14.816
PVP Frag.	0.0	2.8361	101.0	1.0	13.167	251.0	15.475
None (highest LOD), FS	0.0	0.0	0.0	0.0	59.067	117.0	25.951
None (lowest LOD), FS	0.0	0.0	0.0	0.0	38.647	77.0	17.293
orig. BB, FS	0.0	0.88689	6.0	1.0	40.379	78.0	17.600
orig. Pix., FS	0.0	1.0902	118.0	1.0	52.698	383.0	25.956
orig. Frag., FS	0.0	1.1369	108.0	1.0	53.050	380.0	26.625
PVP BB, FS	0.0	0.87049	4.0	1.0	39.951	78.0	17.575
PVP Pix., FS	0.0	0.82295	180.0	1.0	51.207	480.0	25.252
PVP Frag., FS	0.0	0.84344	189.0	1.0	52.349	476.0	26.019

Table 5.2.: Overview of the results in different LOD selection modes; *orig.* references the original cost equation 5.1, *PVP* the adjusted equation 5.2, *BB* is for bounding box, *FS* specifies the rendering with the enhanced fragment shader.

5.4. Comparing graphs of fragments, pixels and costs

The graphs of the fragments and pixels estimated with the lookup table and the bounding box share recurring patterns with some graphs in the costs. The estimated costs in general are a slightly altered version of the estimated pixels and fragments only influenced by the coefficients. The curves of the graphs of the fragments and pixels estimated with the lookup table do not occur within the actual pixels (see Figure 5.3) and fragments (see Figure 5.4). But, they reappear in the diagrams of the costs of the adjusted fragment shader renderings (see Figures 5.5, 5.7 and 5.9). Here, they take the form of the actual costs where all objects in the scene are rendered on the highest LOD.

The curve of the pixels estimated via bounding box (see Figure 5.3 and Figure 5.4) is found again in Figures 5.8, 5.6 and 5.10. The curve appears within the actual costs of the rendering in one single LOD, both LOD 0 and LOD 7, throughout the camera move.

5. Results and Discussion

This shows, that the estimation via bounding box is way more accurate in estimating the actual fragments than the estimation via lookup table. However, it is not necessarily the better tool for estimating the costs.

5.5. Cost estimation with pre-processing

In this section, I summarize the additional costs which arise when the preprocessing step is executed.

5.5.1. Cost

The creation of the lookup tables for the pixels and the fragments need some time. The exact amount of time depends on the number of rotations around the axes. When the number of rotation axes is increased to three, the processing time increases exponentially. The number of objects used in the scene also influences the runtime. Currently, the preprocessing is executed for every object in every level of detail and in every time step. An object can take various shapes with each time step, but currently only one time step for the objects is chosen. The scene itself includes only the aforementioned armadillos. And, therefore, only the preprocessing for one object is executed.

When the actual program runs, the data for each object needs time for being loaded into the memory. The more objects, time steps per object and LODs per object the scene consists of, the more time is needed to read all information and to load them into the RAM.

5.5.2. Memory

The used memory depends, similarly to the preprocessing time, on the number of objects, the number of LODs, time steps, number of samples in the rotation axes and the number of axes. A scene with a single object, eight LODs, one time step, 36 rotation steps in each X- and Y-direction saved as *size_t* would lead to $8 \cdot 36 \cdot 36 \cdot 4 \text{ Bytes} = 41\,472 \text{ Bytes}$. With 100 versions of the object, this increases to about 4 MB.

5.5. Cost estimation with pre-processing

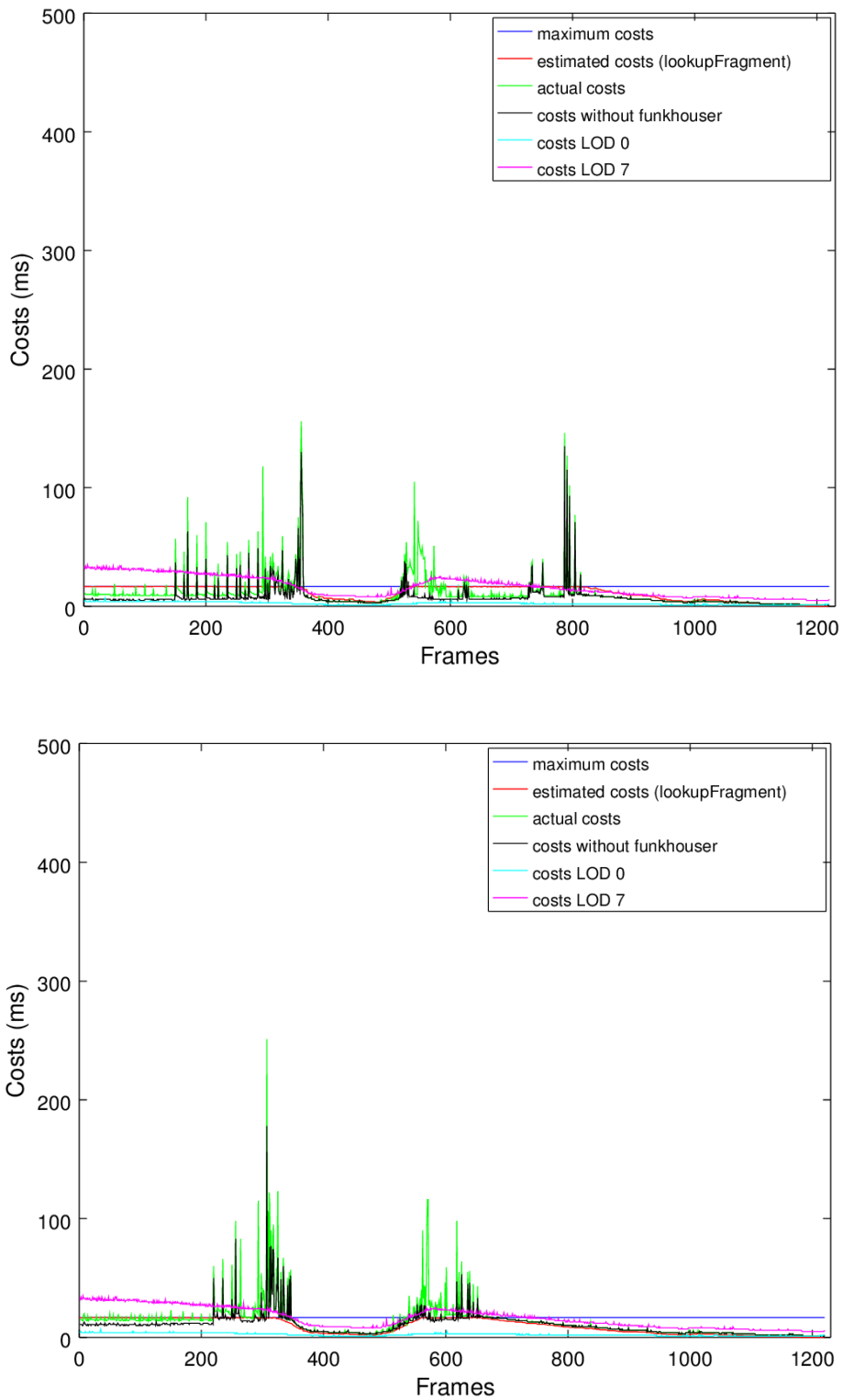


Figure 5.5.: Estimated costs with calculation via fragments over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom.

5. Results and Discussion

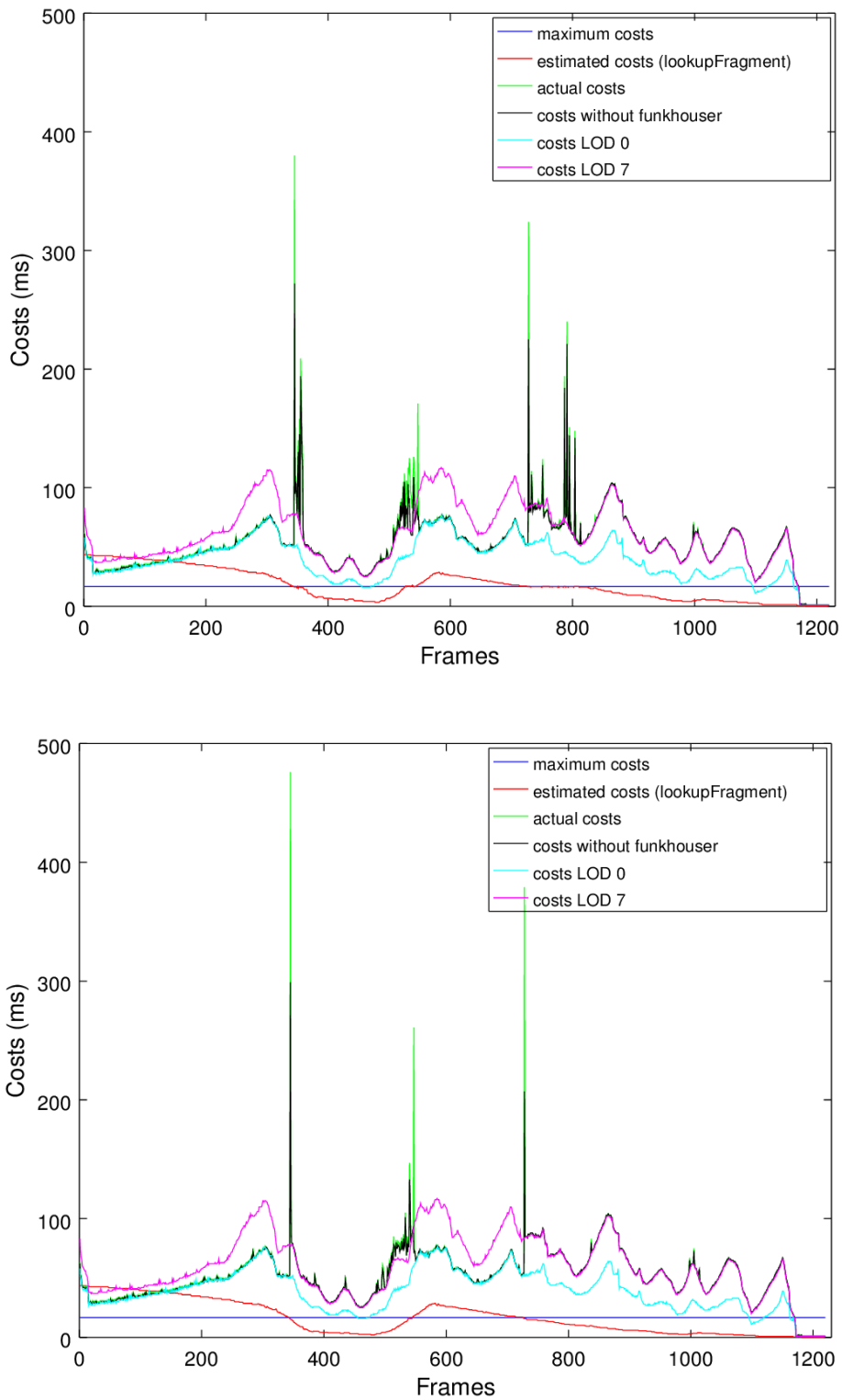


Figure 5.6.: Estimated costs with calculation via fragments over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom. Adjusted fragment shader used.

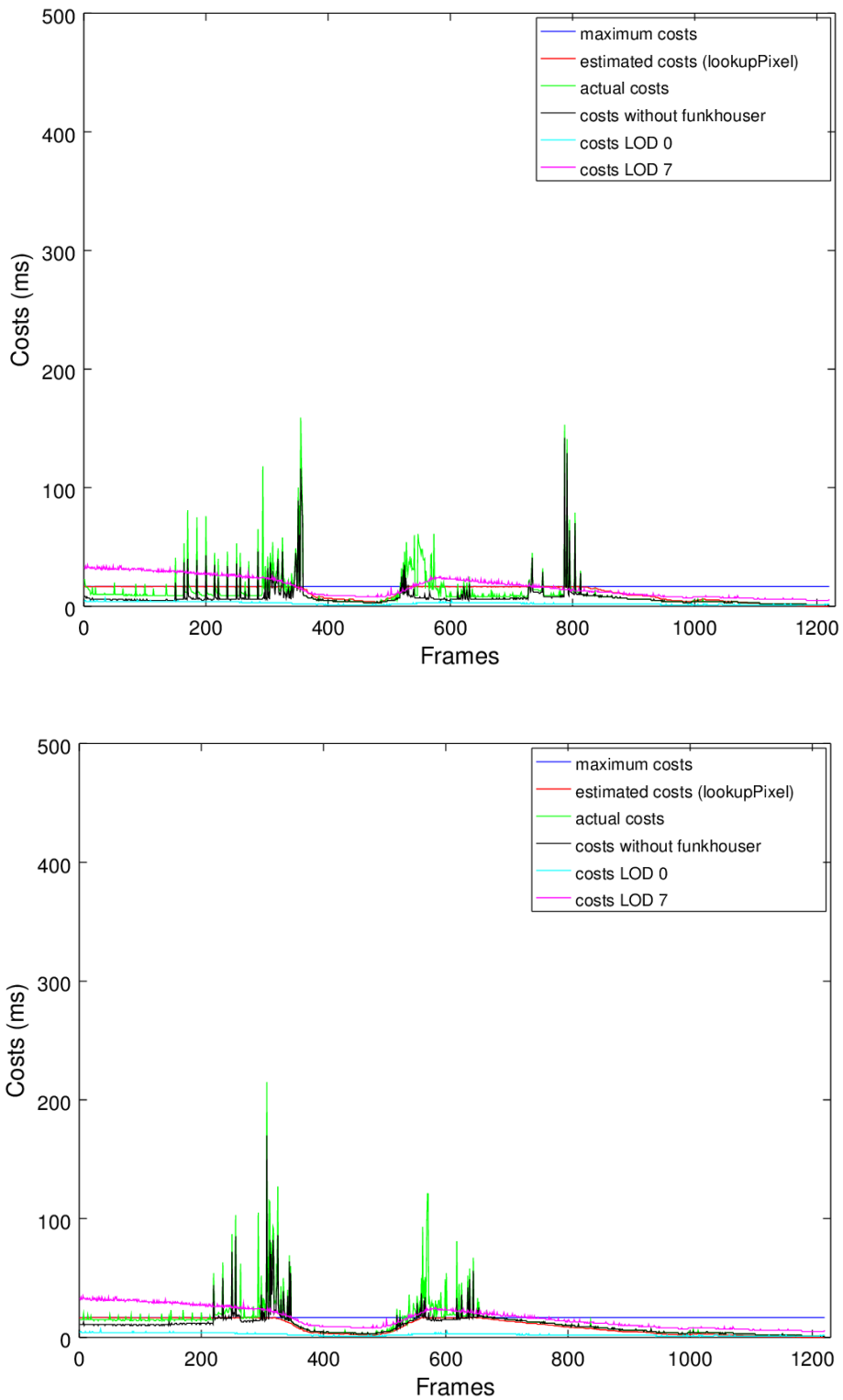


Figure 5.7.: Estimated costs with calculation via pixels over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom.

5. Results and Discussion

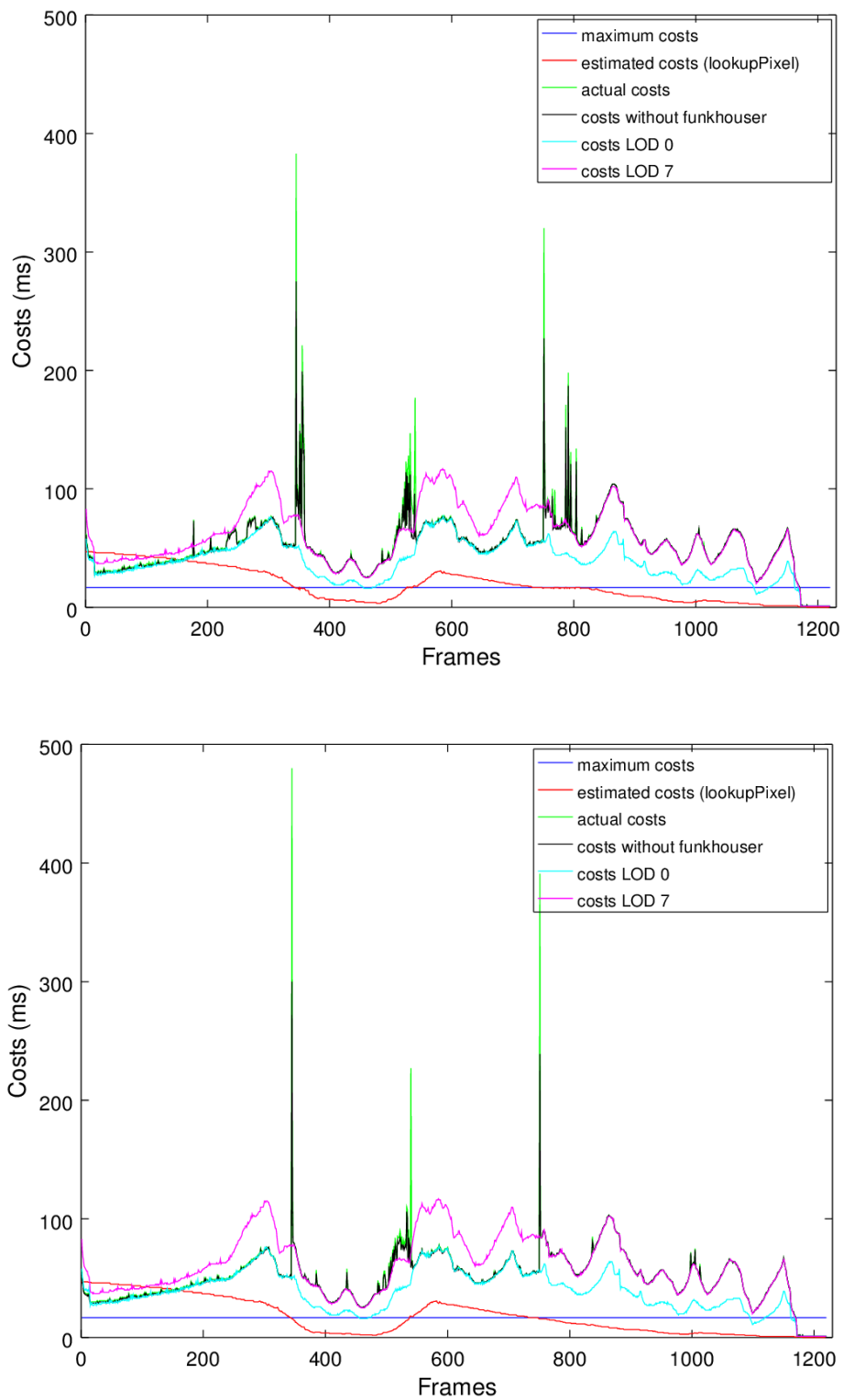


Figure 5.8.: Estimated costs with calculation via pixels over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom. Adjusted fragment shader used.

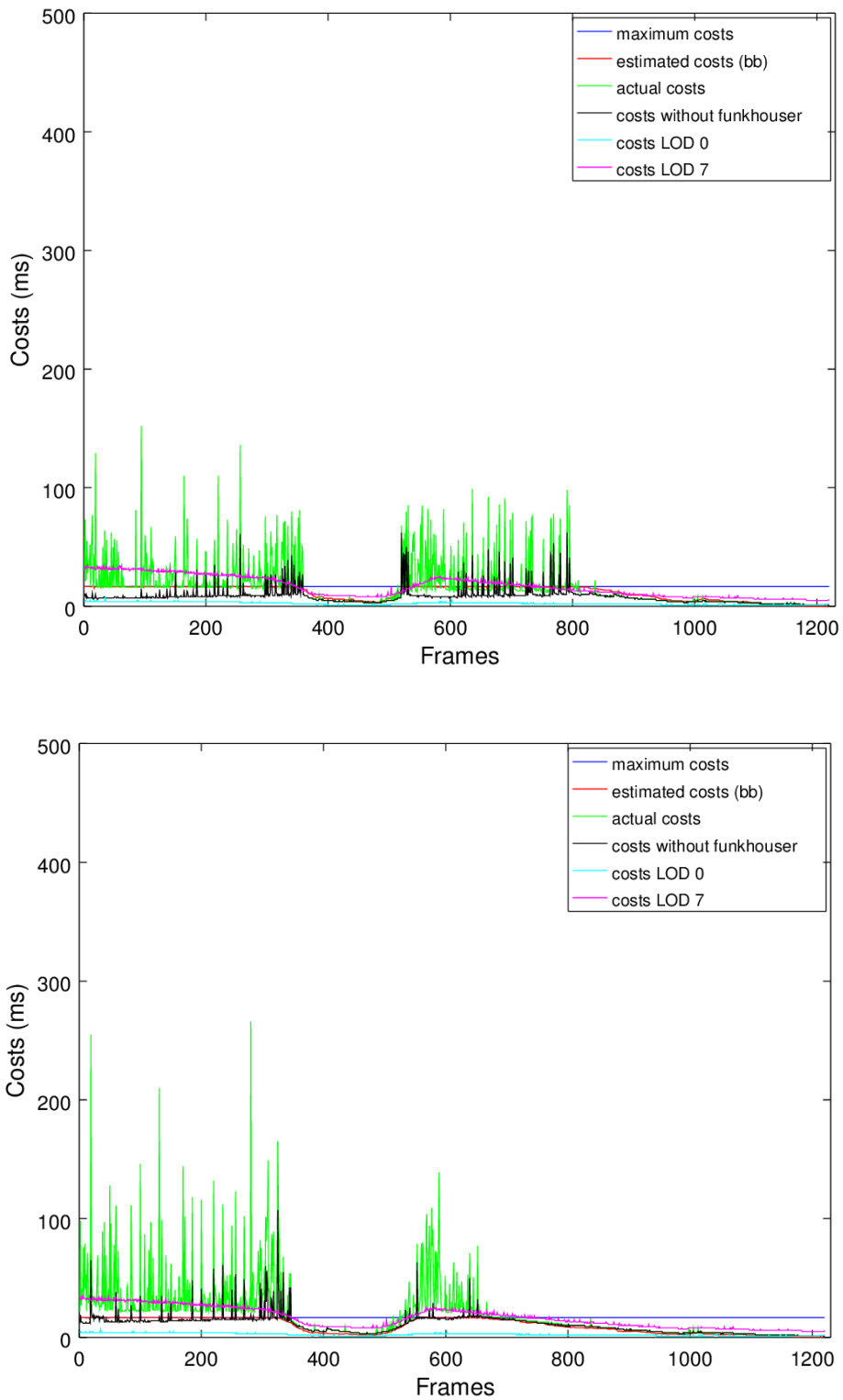


Figure 5.9.: Estimated costs with calculation via bounding box pixels vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom.

5. Results and Discussion

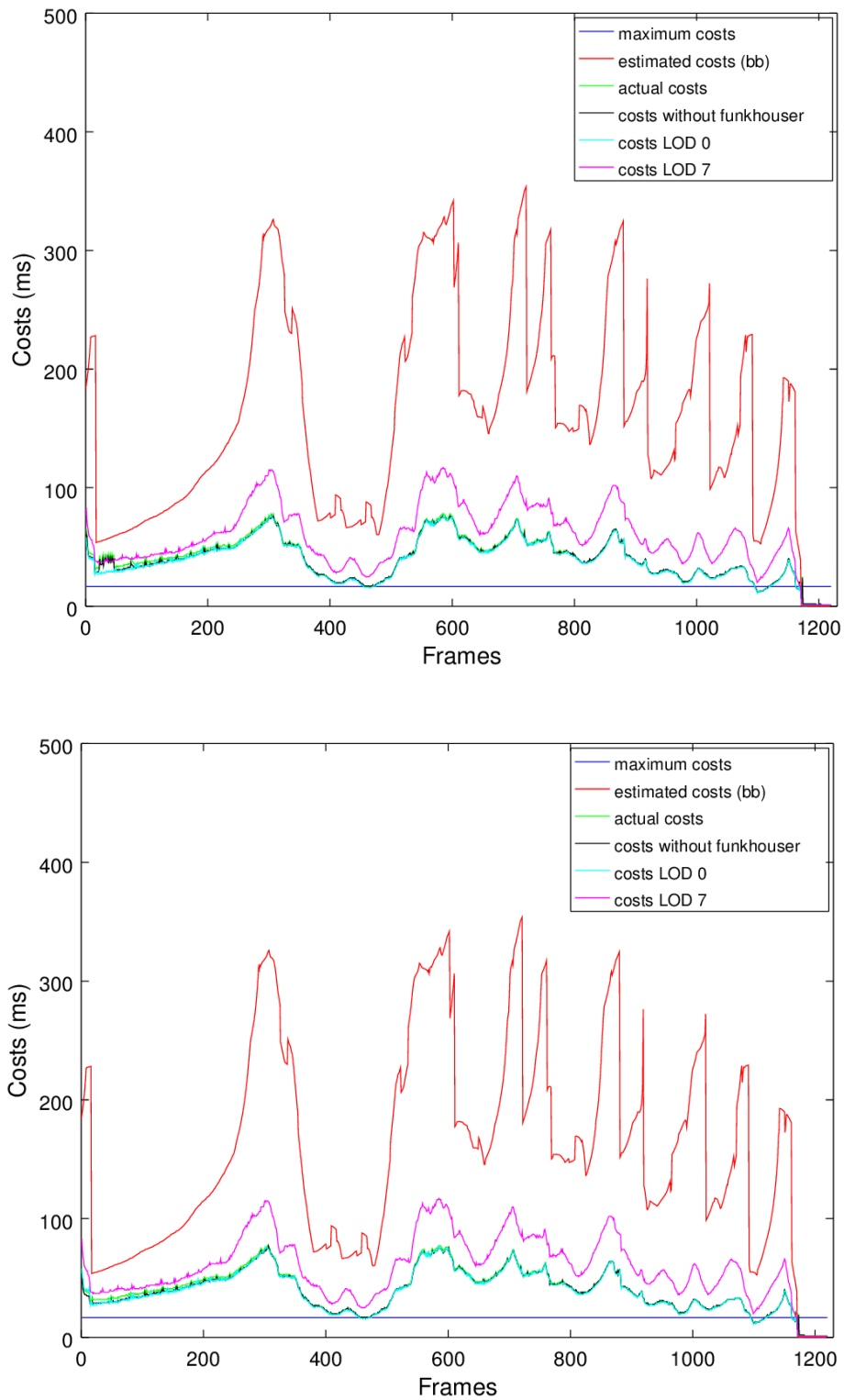


Figure 5.10.: Estimated costs with calculation via bounding box pixels vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom. Adjusted fragment shader used.

6. Conclusion and Future Work

In this chapter, I cover the conclusions I draw from the tests, I executed. And I give a glance on the work which can be done in the future.

When comparing the methods for estimating the fragments, the estimation via bounding box is way more accurate than the estimation via lookup table. However, for estimating the costs, it is not necessarily the better tool. The costs, of rendering the scene on the highest LOD without the adjusted fragment shader (see Figures 5.5, 5.7 and 5.9), show the same form in the curve as the costs estimated via lookup table (see Figure 5.4).

The algorithm of Funkhouser and Séquin is a good start. But it takes not its own execution time into account. Rather than to calculate all changes according to the algorithm, a better idea would be, to measure the time it already took to calculate. This would then be checked against the maximum costs. If the real costs are above this threshold, the algorithm would break off.

Additionally to the costs of the calculation, the algorithm should also include a variable for the time it takes to change the LODs. A starting point might be to double the calculated and/or measured execution time of Funkhouser (as mentioned in the paragraph above). An indication for the doubling is, that the black peaks without the duration of Funkhouser's algorithm in Figure 5.5 through Figure 5.10 are about the half of the height of the total calculation costs. But this needs to be determined empirically.

Another idea is letting the program measure its execution time and to implement a self regulating system. Here, the algorithm itself would only adjust the LODs, if the actual time is above a maximum threshold or under a minimum threshold.

Also the cost calculation still needs some adjusting. Funkhouser and Séquin give the precondition of the determination of coefficients separately while rendering different LODs. When the rendering time is drawn against the vertices, polygons and fragments, the determined coefficients multiplied by the individual values should always add up to the actual cost. This leads to my conclusion, that the equation $cost_{PVP}$ is more accurate than $cost_{original}$.

However, in real world applications especially the time coefficient of the fragments/pixels is also influenced by the respectively rendered LOD. Therefore, a new formula including fragments and vertices and/or polygons is necessary.

The current implementation of Funkhouser and Séquin's algorithm into *screenlib* is still missing a few adaptations, which were already presented in the respective paper. I plan to add a variable for the *semantics* of the objects, a calculation of the *motion blur* and a *focus* tracking system. In 3D projection systems, head tracking is often implemented. With this, it would be feasible to integrate an approach which measures the distance of the object to the view vector.

The current implementation estimates the costs by summing up the estimated fragments or pixels of each visible object. Another set-up for testing in the future is, to correct the estimated pixels from adding up multiple times and to purge the results. The underlying idea is, that the amount of fragments and pixels estimated via lookup table would then align better with the actual amount of fragments, respectively pixels.

6. *Conclusion and Future Work*

Another adjustment would be to save memory and time in the preprocessing step by only creating the lookup table of pixels and fragments for only one LOD. The program would then use the estimation of the fragments and/or pixels for all LODs. This would save memory when loading the object into the RAM. It would also save time, in initialising step of Funkhouser's algorithm, due to the computer loading every lookup table for every object and LOD into the RAM at the beginning of the program's run.

To further reduce memory load, it could also be useful to exploit symmetries. Especially, if an object itself is symmetrical it can be sufficient to save the frames of only one side and to calculate the pixels/fragments for the visualisation on the mirrored side.

7. Acknowledgements

Without some persons and organisations it would not have been possible for me to complete this thesis:

First of all, special thanks to Stanford Computer Graphics Laboratory which maintains freely available models in the Stanford 3D Scanning Repository.

I am also very thankful to Petronella Mann, who provided a lot of motivational support and other input.

And I want to thank my parents and brother, who enabled me to come this far and always supported me on my way.

A very warm thank you also to Mona Kolb, who provided me shelter, food and always an open ear.

Last but not least, I have to emphasise my boyfriend Daniel Kolb, whom I had many theme-related discussions with and who stood solid as a rock.

List of Figures

2.1.	Rendering pipeline under OpenGL 4.3 [SSK13, ch. 1].	4
2.2.	Stanford Armadillo with bounding box in red and fragment shader configured to discard fragments lesser, respectively greater/equal than 0.97 on z-coordinate.	5
2.3.	Sphere with 512 triangles (uniform LOD) [XESV97].	6
2.4.	Sphere with 537 triangles (adaptive LOD) [XESV97].	6
2.5.	Stanford bunny with a decreasing amount of vertices. Triangles from left to right: 69 451, 2 502, 251, 76 [Sch08].	7
2.6.	Stanford bunny with decreasing amount of vertices and smaller appearance, creating a 3D perspective, from left to right [Sch08].	7
2.7.	Rendering times for the market model concept, Funkhouser and Séquin’s algorithm, a rendering on highest LOD and an algorithm determining LODs via a distance algorithm, implemented by Howell et al. [HCSS99].	13
3.1.	Chart of the integration of Funkhouser and Séquin’s algorithm.	15
3.2.	Diagram of the implementation of Funkhouser and Séquin’s algorithm into the existing framework <i>screenLib</i>	16
3.3.	Visualisation of rotation on Z-axis. Approximately, no changes of amount of pixels are recognised.	17
3.4.	Orthographic projection of an icosphere with bounding box in red from three different angles. from left to right: front view, 90° rotation around Y-axis, 90° rotation around Y-axis and by 90° rotation around X-axis.	19
3.5.	Perspective projection of an icosphere with bounding box in red from three different angles; from left to right: front view; 90° rotation around Y-axis; 90° rotation around Y-axis, followed by 90° rotation around X-axis	19
3.6.	The colour scheme used for visualisation of LODs. Dark blue mapped to the lowest LOD, dark red mapped to the highest LOD. The respective color values in RGB from left to right: [69,117,180], [116,173,209], [171,217,233], [224,243,248], [254,224,144], [253,174,97], [244,109,67], [215,48,39].	20
4.1.	Rendering time relative to the amount of polygons.	23
4.2.	Rendering time relative to the amount of vertices.	23
4.3.	Rendering time per amount of fragments.	24
4.4.	Rendering time per amount of fragments with expanded fragment shader.	24
4.5.	Rendering time per amount of pixels.	25
4.6.	Rendering time per amount of pixels with expanded fragment shader.	25
4.7.	Diagram with the construct of one unordered map and two multimaps. The connections via pointer allow to maintain access from one specific object within one map to the same object within the other.	26

List of Figures

4.8.	Use of the trigonometric functions for determining the distance z of the camera to the object. $2a$ equals the diameter of the bounding box of an object. The variable ϕ represents the camera opening angle.	29
4.9.	Use of the intercept theorem for exact camera positioning. The variables w and h are equal to width and height in number of pixels. The relation to $2a$ in world coordinates can thus be computed. The variable a' represents the maximum radius of a bounding sphere of an object which still fits in the screen window.	29
5.1.	Armadillo in different resolutions. Lowest LOD on the top left, highest LOD on the down right.	35
5.2.	Camera route within armadillo scene with the respective camera positions at frames 0, 290, 480, 990, 1220. The ranges indicate changes in camera direction within these frames.	36
5.3.	Total amount of pixels per frame. The number of pixels estimated via bounding box are pictured in green, the number of pixels estimated via lookup table are red and the actual pixels are drawn in blue.	37
5.4.	Total amount of fragments per frame. The amount of fragments estimated via lookup table are pictured in red, the amount of the actual fragments in blue and the amount of pixels estimated with bounding box in green.	38
5.5.	Estimated costs with calculation via fragments over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom.	41
5.6.	Estimated costs with calculation via fragments over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom. Adjusted fragment shader used.	42
5.7.	Estimated costs with calculation via pixels over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom.	43
5.8.	Estimated costs with calculation via pixels over lookup table vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom. Adjusted fragment shader used.	44
5.9.	Estimated costs with calculation via bounding box pixels vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom.	45
5.10.	Estimated costs with calculation via bounding box pixels vs. actual costs in milliseconds per frame. Calculation with the original cost equation 5.1 on the top, the adjusted formula 5.2 on the bottom. Adjusted fragment shader used.	46
A.1.	Armadillo at frame 0. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	62

A.2. Armadillo at frame 290. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	63
A.3. Armadillo at frame 380. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	64
A.4. Armadillo at frame 480. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	65
A.5. Armadillo at frame 590. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	66
A.6. Armadillo at frame 990. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	67
A.7. Armadillo at frame 1020. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	68
A.8. Armadillo at frame 1220. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.	69

List of Tables

2.1.	Abbreviations used by Funkhouser and Séquin.	8
2.2.	Abbreviations used by Gobbetti and Bouvier	11
3.1.	Excerpt of a lookup table with placeholder fragment counts per angle.	17
4.1.	Coefficients determined via rendering of exemplary objects for regular Phong shading and the expanded fragment shader.	22
4.2.	Excerpt of a an actual lookup table with fragment counts per angle.	31
4.3.	Contingency table for computing quotients of fragment counts.	31
5.1.	Table of number of pixels of a sphere with and without bounding box. Those are calculated via screenshot and using GIMP 2.8.10 [gim17] and its histogram tool.	34
5.2.	Overview of the results in different LOD selection modes; <i>orig.</i> references the original cost equation 5.1, <i>PVP</i> the adjusted equation 5.2, <i>BB</i> is for bounding box, <i>FS</i> specifies the rendering with the enhanced fragment shader.	39

Bibliography

- [AL99] Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 307–316, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.
- [Are17] LLC. ArenaNet. Guild wars 2. <https://www.guildwars2.com/en/>, 2017. [Online; accessed 2017-07-06].
- [ata01] atan2(3) - linux man page. <https://linux.die.net/man/3/atan2>, 2001. [Online, accessed 2017-01-10].
- [BH13] Cynthia Brewer and Mark Harrower. Colorbrewer 2.0. <http://colorbrewer2.org/>, 2013. [Online; accessed 2017-04-28].
- [Bla90] E. H. Blake. The natural flow of perspective: Reformulating perspective projection for computer animation. *Leonardo*, 23(4):401–409, 1990.
- [Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, October 1976.
- [EMB01] Carl Erikson, Dinesh Manocha, and William V. Baxter, III. Hlods for faster display of large static and dynamic environments. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 111–120, New York, NY, USA, 2001. ACM.
- [Fou17] Blender Foundation. Blender. <https://www.blender.org/>, 2017. [Online; accessed 2017-04-05].
- [FS93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 247–254, New York, NY, USA, 1993. ACM.
- [GB99] Enrico Gobbetti and Eric Bouvier. Time-critical multiresolution scene rendering. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, VIS '99, pages 123–130, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [gim17] Gnu image manipulation program (gimp). <https://www.gimp.org/downloads/>, 2017. [Online; accessed 2017-04-04].
- [HB03] Mark Harrower and Cynthia A. Brewer. Colorbrewer.org: An online tool for selecting colour schemes for maps. *The Cartographic Journal*, 40(1):27–37, 2003.

- [HCSS99] J. Howell, Y. Chrysanthou, A. Steed, and M. Slater. A market model for level of detail control. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST '99*, pages 96–103, New York, NY, USA, 1999. ACM.
- [Ho11] Nghia Ho. Decomposing and composing a 3×3 rotation matrix. http://nghiaho.com/?page_id=846, 2011. [Online; accessed 2017-01-02].
- [IHTI78] Toshihide Ibaraki, Toshiharu Hasegawa, Katsumi Teranaka, and Jiro Iwase. The multiple choice knapsack problem. *J. Oper. Res. Soc. Japan*, 21:59–94, 1978.
- [Jar73] R.A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18 – 21, 1973.
- [KS99] James T. Klosowski and Cláudio T. Silva. Rendering on a budget: A framework for time-critical rendering. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years, VIS '99*, pages 115–122, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [Lin03] Peter Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics, I3D '03*, pages 93–102, New York, NY, USA, 2003. ACM.
- [NW98] Shaun Nirenstein and Simon Winberg. Algorithms. <https://webcache.googleusercontent.com/search?q=cache:w9JiWgkB0vQJ:https://people.cs.uct.ac.za/~snirenst/Documents/LodReport/node6.html>, 1998. [Online; accessed 2016-09-27].
- [PPCT11] Chao Peng, Seung In Park, Yong Cao, and Jie Tian. *A Real-Time System for Crowd Rendering: Parallel LOD and Texture-Preserving Approach on GPU*, pages 27–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [Sch08] Roman Schulz. Polygon reducer 2.0 - part 2: Reducing level of detail. <http://polygon-reducer.pc-guru.cz/reducing-level-of-detail>, 2008. [Online; accessed 2017-03-29].
- [SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, January 1974.
- [SSKLLK13] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.
- [sta14] The stanford 3d scanning repository. <http://graphics.stanford.edu/data/3Dscanrep/>, 2014. [Online; accessed 2017-04-05].
- [Tea17] Assimp Development Team. Open asset import library. <http://assimp.sourceforge.net/>, 2017. [Online; accessed 2017-04-04].
- [VM02] Gokul Varadhan and Dinesh Manocha. Out-of-core rendering of massive geometric environments. In *Visualization, 2002. VIS 2002. IEEE*, pages 69–76, Nov 2002.

- [XESV97] Julie.C. Xia, Jihad El-Sana, and Amitabh Varshney. Adaptive real-time level-of-detail based rendering for polygonal models. *Visualization and Computer Graphics, IEEE Transactions on*, 3(2):171–183, Apr 1997.
- [Zac16] Gabriel Zachmann. Virtual reality & physically-based simulation - techniques for real-time rendering. University Lecture, http://cgvr.informatik.uni-bremen.de/teaching/vr_1617/fohlen/05%20-%20Real-time%20rendering.pdf, 2016. [Online; accessed 2017-03-15].

A. Appendix

$$\text{atan2}(y, x) = \begin{cases} +\pi & \text{if } x < 0 \text{ and } y = +0, \\ -\pi & \text{if } x < 0 \text{ and } y = -0, \\ +0 & \text{if } x > 0 \text{ and } y = +0, \\ -0 & \text{if } x > 0 \text{ and } y = -0, \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0, \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0, \\ NaN & \text{if } x = NaN, \\ NaN & \text{if } y = NaN, \\ +\pi & \text{if } x = -0 \text{ and } y = +0, \\ -\pi & \text{if } x = -0 \text{ and } y = -0, \\ +0 & \text{if } x = +0 \text{ and } y = +0, \\ -0 & \text{if } x = +0 \text{ and } y = -0, \\ +\pi & \text{if } x = -\infty \text{ and } y > 0, \\ -\pi & \text{if } x = -\infty \text{ and } y < 0, \\ +0 & \text{if } x = \infty \text{ and } y > 0, \\ -0 & \text{if } x = \infty \text{ and } y < 0, \\ +\frac{\pi}{2} & \text{if } x = \{R\} \text{ and } y = +\infty, \\ -\frac{\pi}{2} & \text{if } x = \{R\} \text{ and } y = -\infty, \\ +\frac{3\pi}{4} & \text{if } x = -\infty \text{ and } y = +\infty, \\ -\frac{3\pi}{4} & \text{if } x = -\infty \text{ and } y = -\infty, \\ +\frac{\pi}{4} & \text{if } x = +\infty \text{ and } y = +\infty, \\ -\frac{\pi}{4} & \text{if } x = +\infty \text{ and } y = -\infty. \end{cases} \quad (\text{A.1})$$

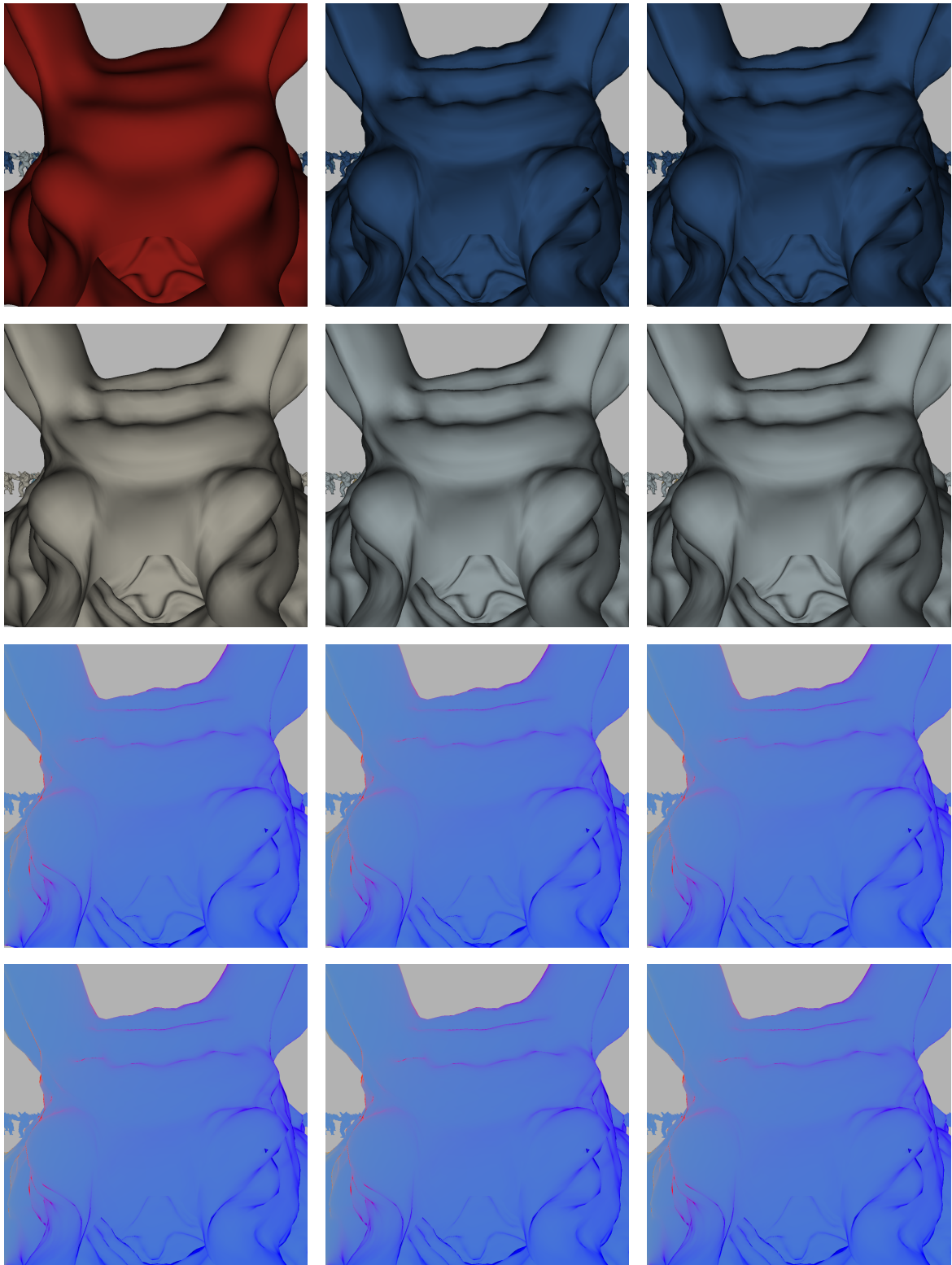


Figure A.1.: Armadillo at frame 0. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.

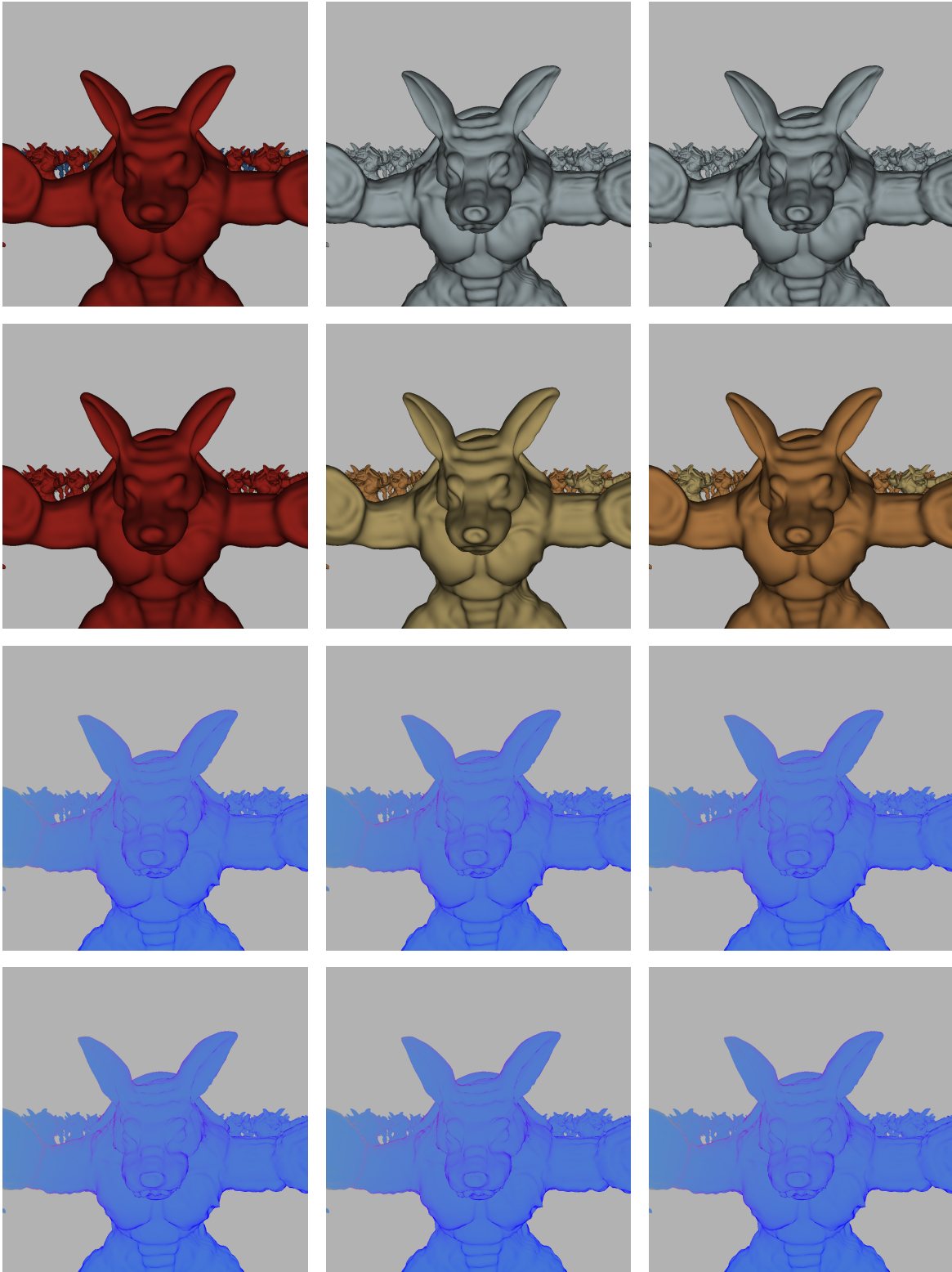


Figure A.2.: Armadillo at frame 290. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.

A. Appendix

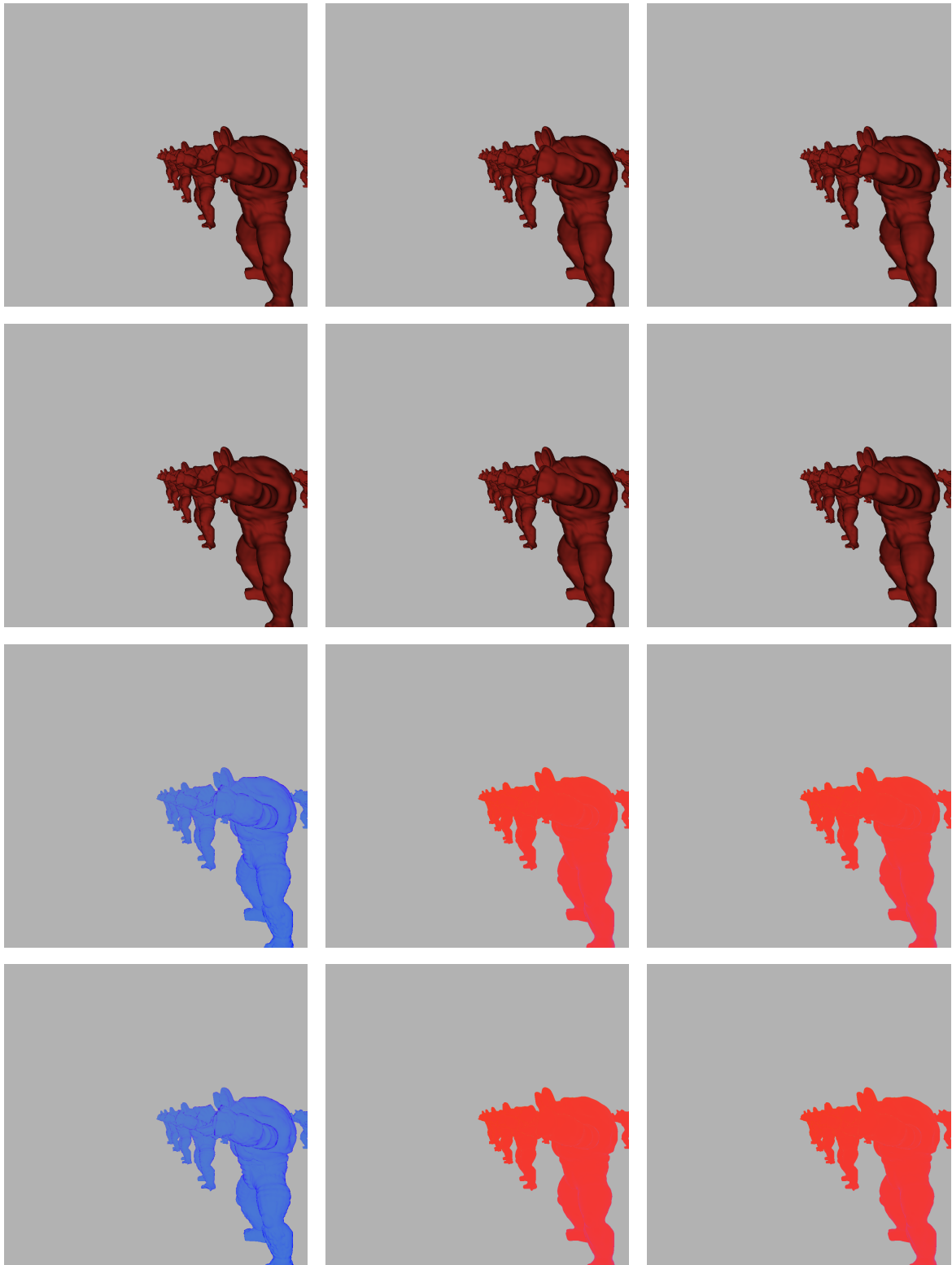


Figure A.3.: Armadillo at frame 380. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.

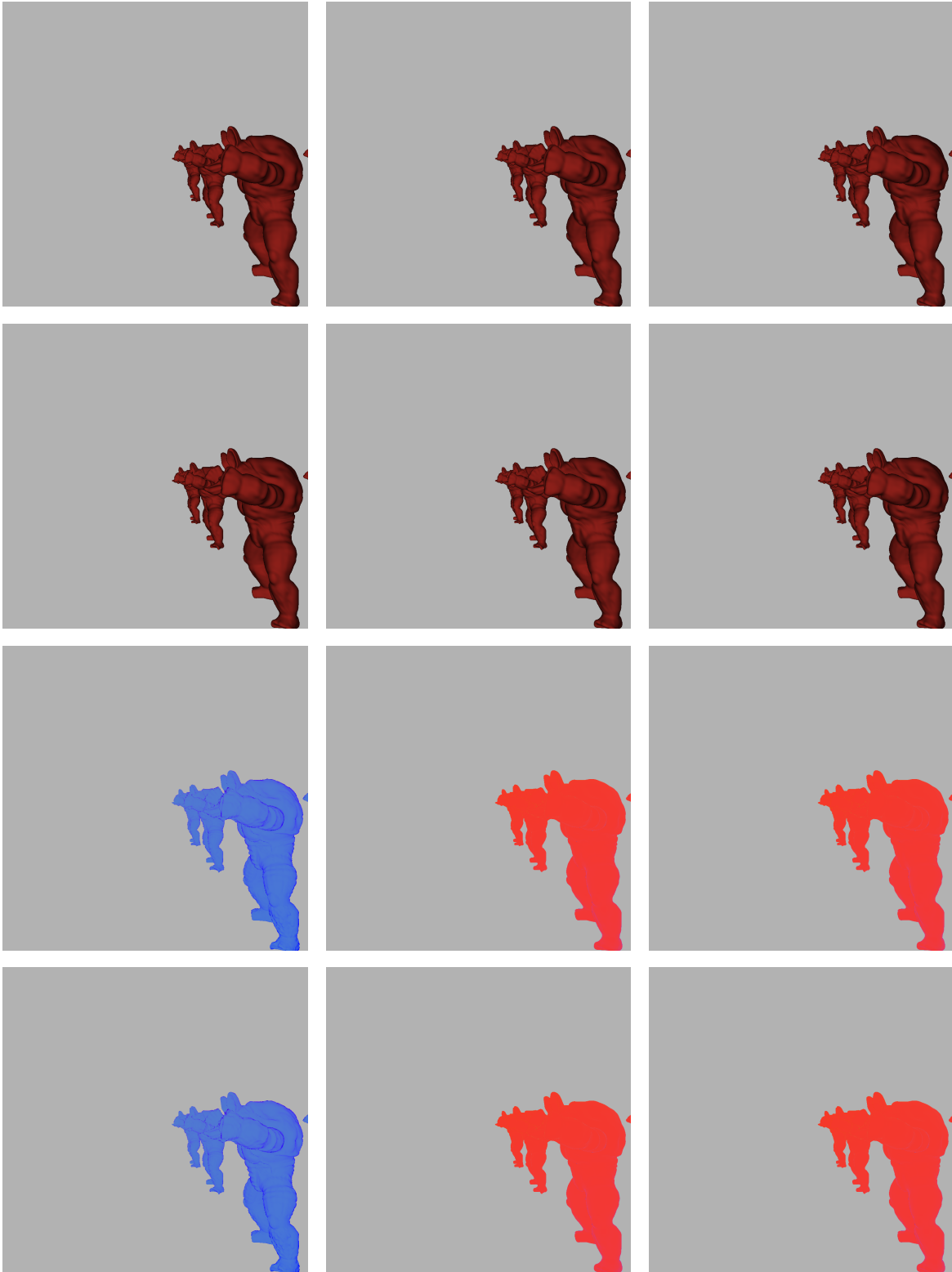


Figure A.4.: Armadillo at frame 480. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.

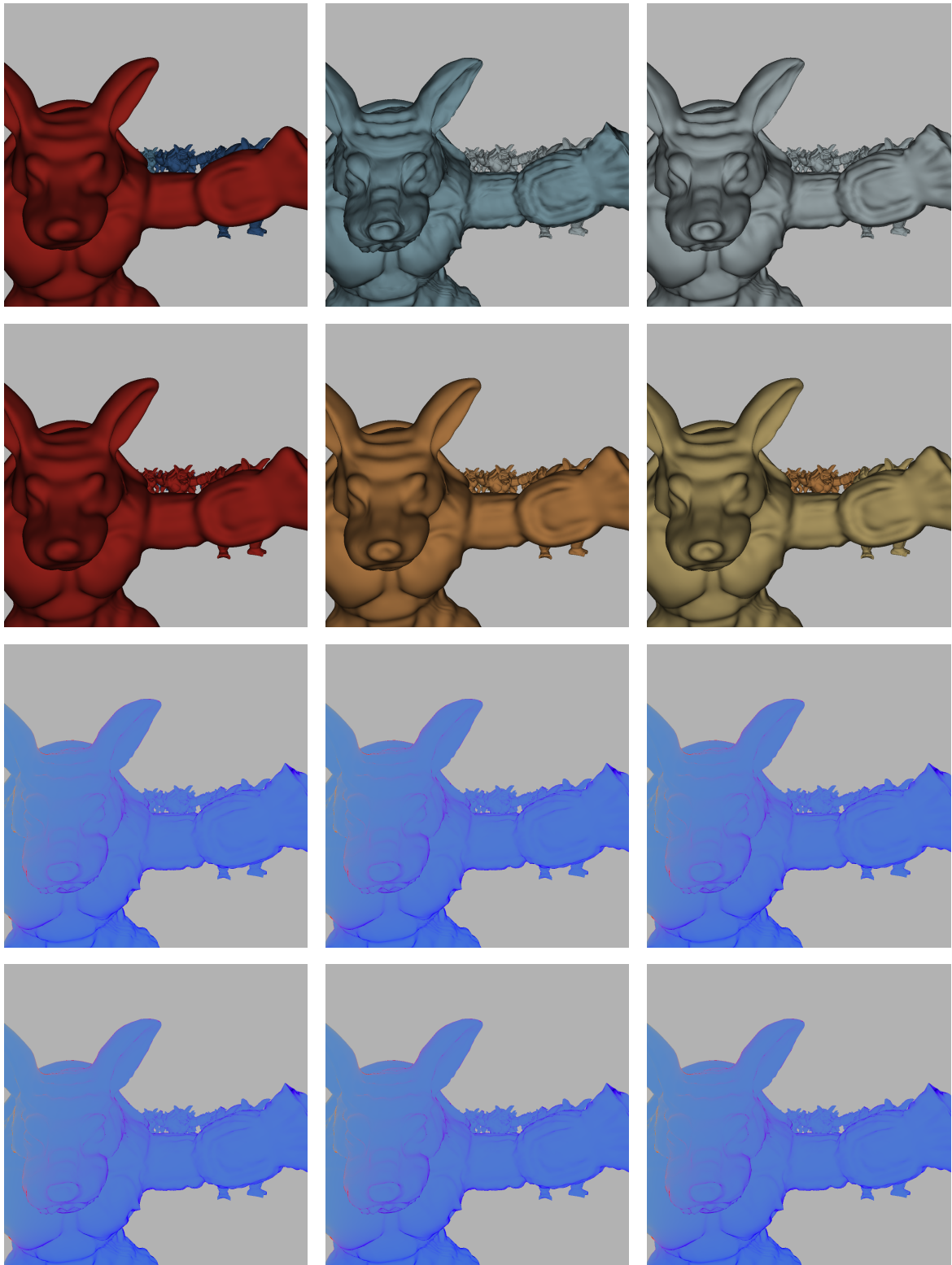


Figure A.5.: Armadillo at frame 590. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.

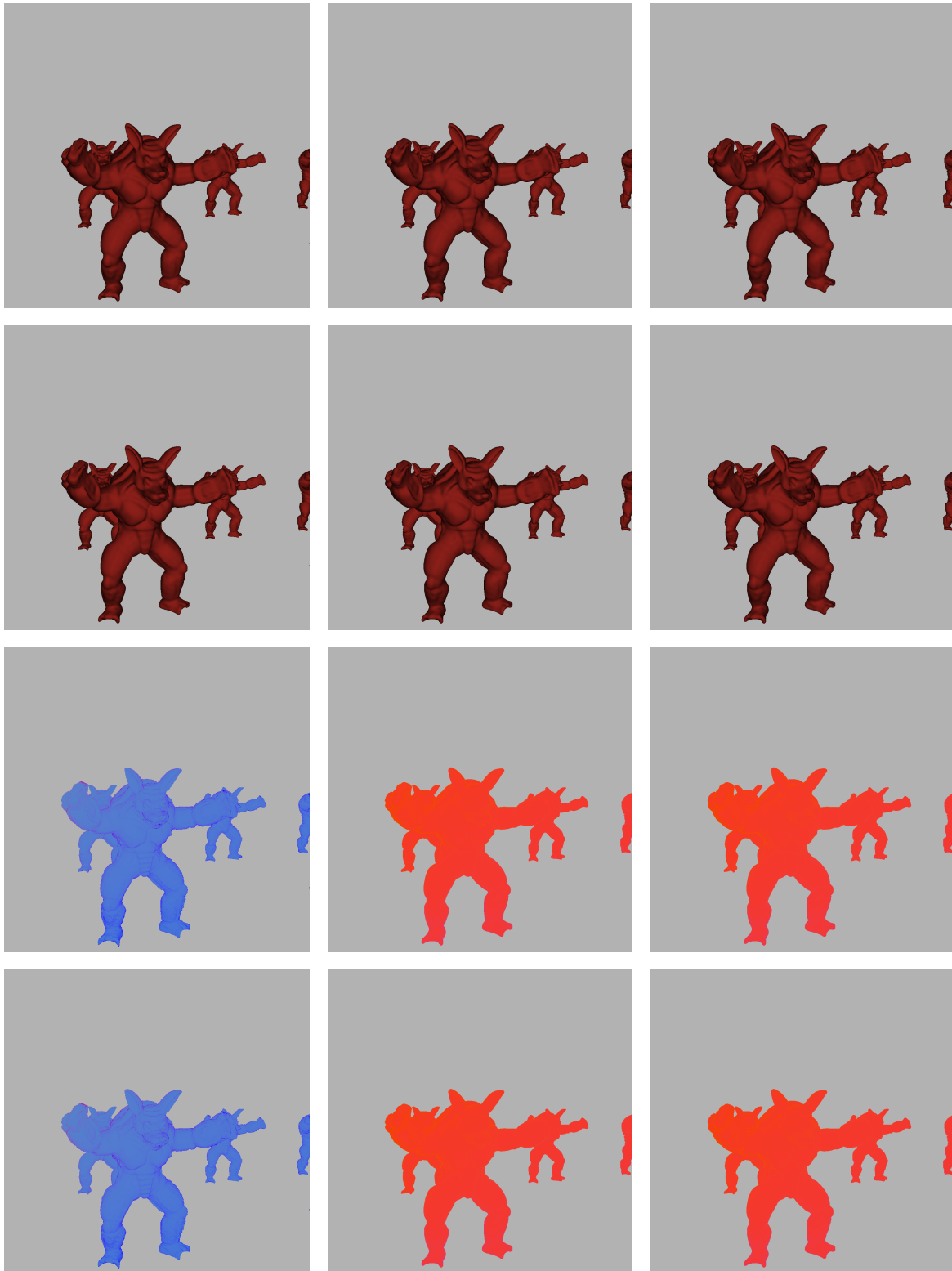


Figure A.6.: Armadillo at frame 990. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.

A. Appendix

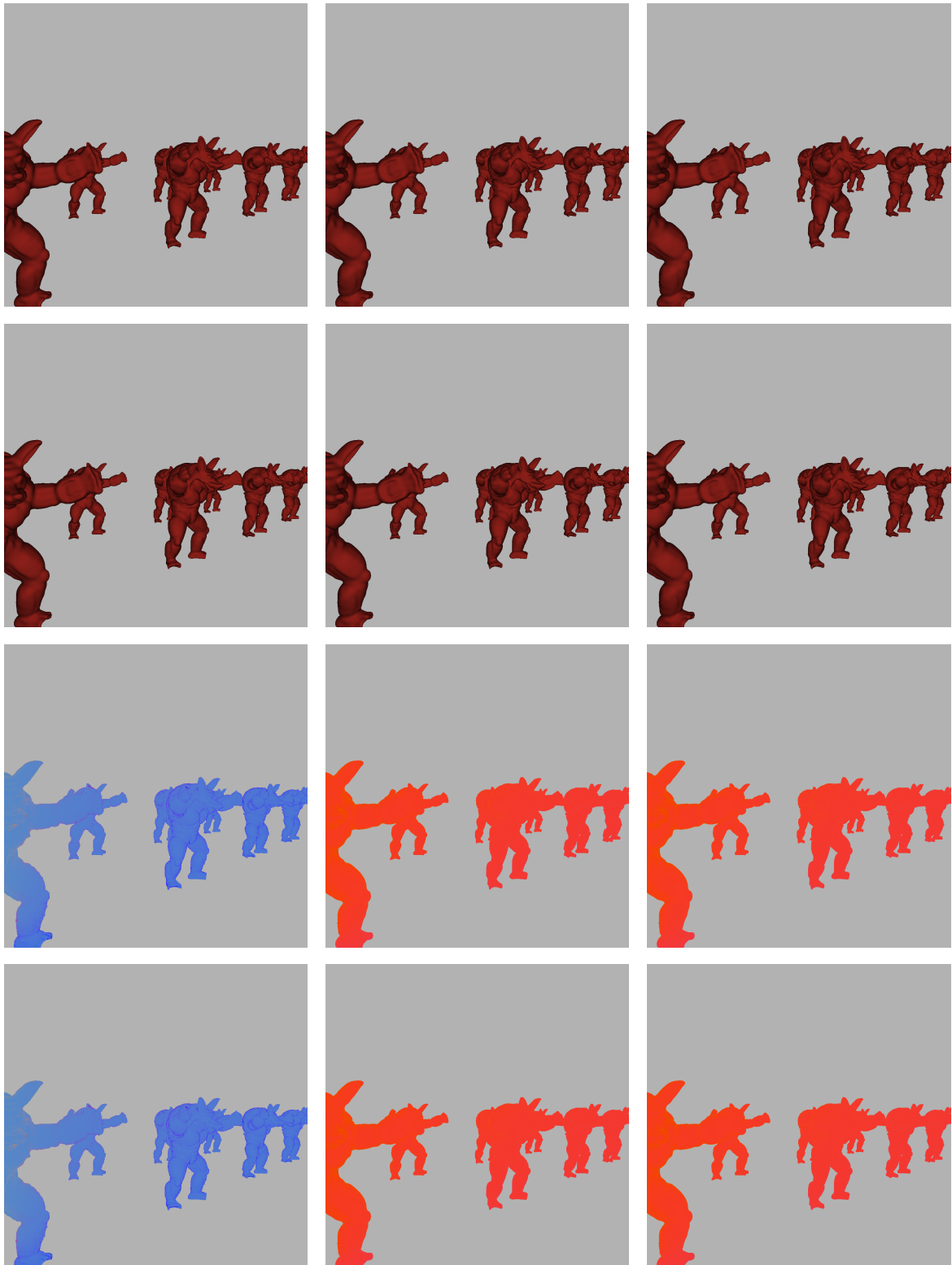


Figure A.7.: Armadillo at frame 1020. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.

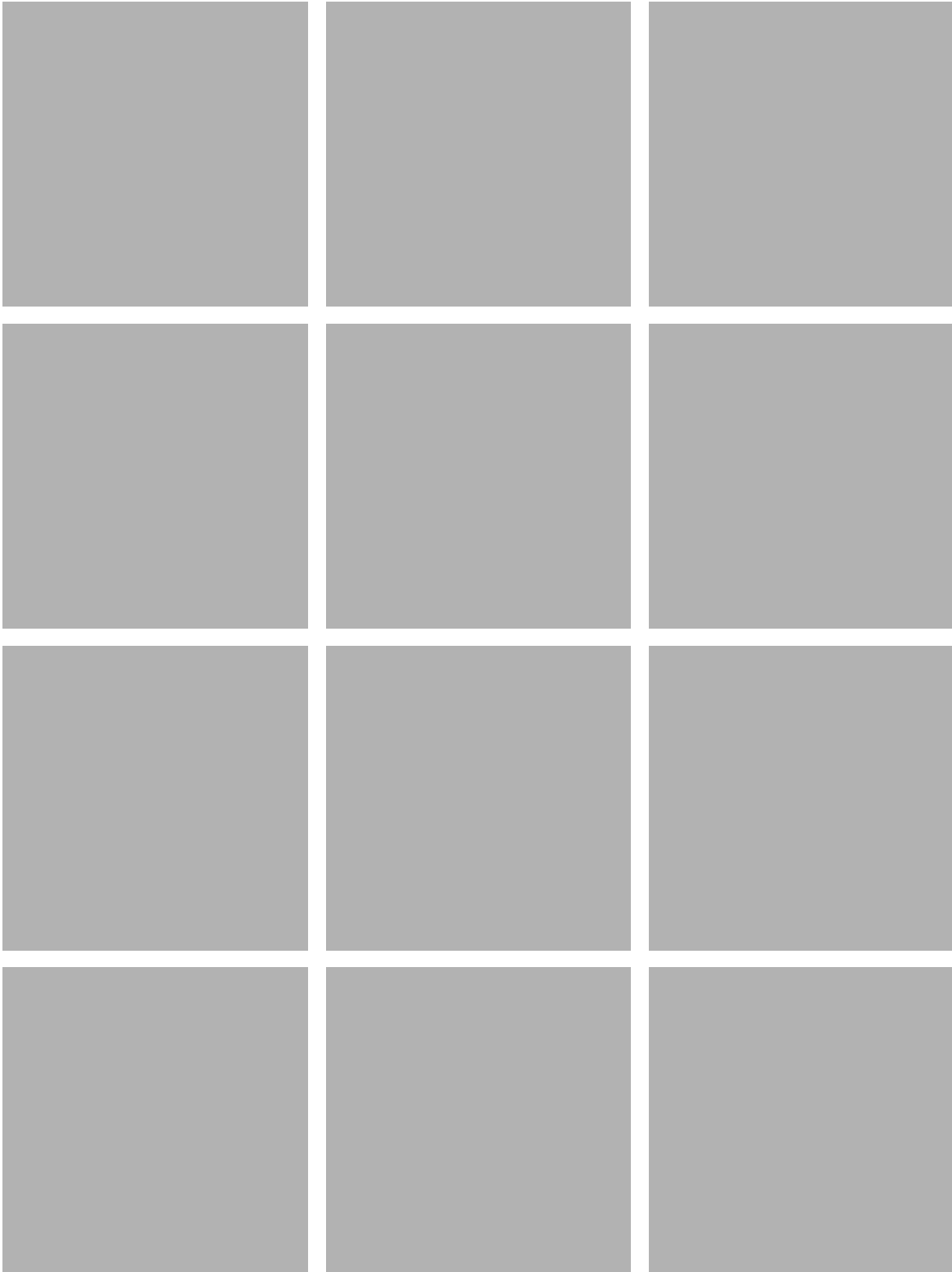


Figure A.8.: Armadillo at frame 1220. Columns: Bounding box, lookup table fragments and lookup table pixels estimation; rows: original cost formula, PVP formula, original cost formula with adjusted fragment shader, PVP formula with adjusted fragment shader.