GPU Shader für geometrische Grundprimitive

Großer Beleg
im Fachgebiet Softwaretechnik
Prof. Dr. Stefan Gumhold
Institut für Software- und Multimediatechnik
Lehrstuhl für Computergrafik und Visualisierung
Fakultät Informatik
Technische Universität Dresden

Patrick Brausewetter
15. Juni 2008
# Contents

1. Motivation 6

2. Related Work 7

3. Overall Approach 13
  3.1. Overview ........................................ 13
  3.2. General Properties of the Technique .................... 15
    3.2.1. Usage ........................................ 15
    3.2.2. Integration into OpenGL .......................... 16
  3.3. Vertex Shaders .................................... 17
    3.3.1. Silhouette ................................... 17
    3.3.2. Acceleration ................................. 18
  3.4. Fragment Shaders .................................. 18
    3.4.1. Intersection Shaders ........................... 18
    3.4.2. Illumination Shader ............................ 21
    3.4.3. Deferred Shading ................................ 21

4. Special Shaders 25
  4.1. Sphere ............................................ 25
    4.1.1. Vertex Shader ................................ 25
    4.1.2. Intersection Shader ................................ 27
      a. Explicit Solution .................................. 27
      b. Incremental Solution ................................. 29
      c. Normal Computation ................................ 31
  4.2. Cylinder .......................................... 32
    4.2.1. Vertex Shader ................................ 32
      a. Case 1 (inside) ..................................... 32
      b. Case 2 (outside) .................................... 34
    4.2.2. Intersection Shader ............................... 36
      a. Explicit Solution ................................ 37
      b. Incremental Solution ............................... 38
      c. Normal Computation ................................ 39
  4.3. Cone .............................................. 40
    4.3.1. Vertex Shader ................................ 40
      a. Case 1 (inside) ..................................... 40
      b. Case 2 (outside above) .............................. 42
      c. Case 3 (outside below) .............................. 43
      d. Case 4 (outside front, low) .......................... 45
      e. Case 5 (outside front) .............................. 49
    4.3.2. Intersection Shader ............................... 53
      a. Solution (explicit) ................................ 53
      b. Normal Computation ................................ 56
5. Framework
  5.1. GeometricRenderer ............................................. 58
  5.2. Camera .......................................................... 59
  5.3. ShaderLibrary ................................................... 59
  5.4. DeferredShader .................................................. 60
  5.5. Scene ............................................................. 60

6. Performance Analysis .............................................. 61
  6.1. General Durations ............................................... 61
  6.2. Silhouette Determination ...................................... 61
  6.3. Intersection Computation .................................... 62
  6.4. Deferred Shading ............................................... 63
  6.5. Rendering time ................................................ 63
  6.6. Shader-Tessellation-Comparison ............................ 64

7. Limitations and Future Prospects ................................ 66
  7.1. Deferred Shading ............................................... 66
  7.2. Backface Culling ............................................... 66
  7.3. Texturing ....................................................... 66
  7.4. More Primitives ................................................ 66
  7.5. Conclusion ..................................................... 67

List of Figures ......................................................... 68

List of Tables .......................................................... 69

Bibliography ............................................................ 70
Erklärung

Hiermit erkläre ich, Patrick Brausewetter, den vorliegenden Großen Beleg zum Thema:

**GPU Shader für geometrische Grundprimitive**

selbstständig und ausschließlich unter Verwendung der angegebenen Literatur- und Informationsquellen verfasst zu haben.

Dresden, den

Patrick Brausewetter
### Mathematical Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{a} = \begin{pmatrix} a_x \ a_y \ a_z \ a_w \end{pmatrix}$</td>
<td>$= (a_x, a_y, a_z, a_w)^T$</td>
</tr>
<tr>
<td>$A = \begin{pmatrix} A_x \ A_y \ A_z \end{pmatrix}$</td>
<td>$= (A_x, A_y, A_z)^T$</td>
</tr>
<tr>
<td>$</td>
<td>\vec{a}</td>
</tr>
<tr>
<td>$\vec{a} - \vec{b}$</td>
<td></td>
</tr>
<tr>
<td>$\vec{x} \cdot a$</td>
<td></td>
</tr>
<tr>
<td>$M = \begin{pmatrix} m_{11} &amp; m_{12} &amp; m_{13} &amp; m_{14} \ m_{21} &amp; m_{22} &amp; m_{23} &amp; m_{24} \ m_{31} &amp; m_{32} &amp; m_{33} &amp; m_{34} \ m_{41} &amp; m_{42} &amp; m_{43} &amp; m_{44} \end{pmatrix}$</td>
<td></td>
</tr>
<tr>
<td>$M \cdot \vec{x}$</td>
<td></td>
</tr>
<tr>
<td>$\vec{x} \times \vec{y}$</td>
<td></td>
</tr>
<tr>
<td>$[a, b]$</td>
<td></td>
</tr>
<tr>
<td>$A \equiv B$</td>
<td></td>
</tr>
</tbody>
</table>
1. Motivation

Basic Geometric Primitives are widely used in Computer Graphics. They are used for modeling, especially in CAD systems, where complex objects are built up from a variety of primitives (points, lines, ellipsoids etc.).

In [13] a method for segmenting complex objects in geometric objects for better recognition of the environment by robots is introduced. Wu also references Biederman who shows in his theory of Recognition-By-Components ([1]) such a segmentation of complex objects for faster identification of objects. In [9] a similar approach is shown where Patel restricts to the so-called “SymGeons” (Symmetrical Geometric Icons) as basic primitives, because they correlate with the human assessment of sensed complexity. Complex foreground objects are modeled out of nine primitives, amongst others spheroid, cylinder and cone. Another advantage of using SymGeons is that the visual complexity of objects can be measured.

They are also common in the area of visualization. Often physical quantities are visualized through many basic primitives. For displaying vector fields many spheres are drawn along the vector directions. In the area of visualizing molecules the single atoms the complex molecules consist of are drawn in different colors.

For all these applications it would be nice to be able to draw such basic geometric primitives in a very fast manner. The normal way of tessellating them into a smooth polygonal mesh leads to very high counts of polygons where often one polygon only covers one pixel because of the high distance to the viewer. In this work I implemented a library working on top of OpenGL that avoids this problem utilizing the benefits of GLSL shaders. The shaders are optimized for performance but are also able to mimic the normal OpenGL behavior (especially lighting).

OpenGL is a widely used graphics library, that is platform-independent and is subject to a standardization process performed by an industrial consortium. Many applications using it could benefit from this work because the covered primitives are also used for many other areas, like e.g. counters in a chess game, the usage is very simple and it offers only a few drawbacks. It can also be the beginning of a growing framework in which many more primitives can be used: Tori, simplexes, NURBS surfaces (like in [4]) and many others.
2. Related Work

My work is based on [3], where Gumhold presents a way of drawing ellipsoids by splatting their silhouette and then rendering this silhouette with a special fragment shader in GLSL. The basic principle there is the same I used for my work: First the outline is calculated and a quadrilateral is built up from that. Then for every fragment in this quadrilateral a ray-primitive (here ray-sphere) collision detection test is made. If there is no intersection, the fragment (pixel) is discarded, otherwise the real position on the primitive is computed and out of this also the normal. With this two values a lighting calculation is done and the real depth-value is applied to the fragment. To achieve a good performance, the calculations in the fragment shader are done incrementally, i.e. some quantities are pre-calculated in the vertex shader and with their interpolated values the real calculation is done.

![Figure 2.1.: Screenshot from [3] (tensor field visualized by large number of ellipsoids)](image)

In [11] the authors describe a similar approach for quadratic surfaces. Patches of quadrics are rendere by splatting the visible area of a bounding area that defines the area of interest of the quadric. The approach is applied to SLIM surfaces ([8]), where the bounding area is a sphere which is splatted to a quadrilateral. There the first few intersection points for every ray are blended to get the real point that is rendered. Also tetrahedral meshes and bilinear qudrilaterals are considered. The front facing triangles of their bounding tetrahedra are raytraced there to compute the actual points. The quadric itself is provided as a square matrix of the dimension 4 to the shader, which is very flexible, but also inefficient in
comparison to specialised shaders for various geometric primitives.

![Figure 2.2: Screenshot from [11] (dragon SLIM model with different shaders)](image)

A general solution for rendering implicit surfaces is introduced by Loop and Blinn in [6]. The primitives they use are algebraic surfaces defined by trivariate Bézier tetrahedra. Because of the use of algebraic methods for finding the roots the maximum degree of the surfaces is 4. The surfaces are only rendered inside a bounding tetrahedron. The rendering of the tetrahedron is done in 2 phases: First the visible non-overlapping triangles creating the silhouette of the tetrahedron are determined in screen space on the CPU, then these triangles are rendered. For every pixel emerging from the rasterization the ray-intersection with the surface is computed in screen space with the help of the Bernstein basis. At last the roots have to be checked against the tetrahedron which is done by testing if they lie in a given interval that is interpolated from the 3 intervals calculated at the vertices of the triangle and the tangent plane is determined from which the normal is deduced for the lighting calculation.

![Figure 2.3: Screenshot from [6] (torus clipped by bounding tetrahedron)](image)

Klein and Ertl present in [5] an illustration technique for magnetic field lines. The visualization is done through ellipsoidal shaped particles that consist of ferromagnetic materials. The particle simulation is computed in a cluster of several PCs. For each ellipsoid only a single OpenGL point is rendered, so only one vertex has to be sent to the vertex shader where the actual screen size of the ellipsoid is computed and then set as point size for this vertex. For every fragment of this point a ray-sphere intersection...
is calculated in object coordinates and the fragment is either discarded or lighted in the usual way. There are two disadvantages of this method: First of all a point in OpenGL is always square, so the approximation of the ellipsoid’s outline is not so good, resulting in a higher fragment count. Secondly no incremental raycasting can be done, because only one vertex is available for each ellipsoid, which leads to a lower performance.

![Figure 2.4: Screenshot from [5] (bar magnet illustrated by 6,000 particles)](image)

Reina and Ertl show in [10] an analog method for the visualization of molecular glyphs. One glyph consists of two molecules (displayed as spheres) and one symbolic bar magnet (a cylinder). Again they render only one ‘fat point’ in an appropriate size containing the whole glyph and then raytrace every fragment to decide whether the fragment is on a sphere or on the cylinder or neither of them, whereas it is discarded. With the right position a phong shading is done to achieve high visual quality. Their work would benefit, as they stated themselves, even more from a closer approximation of the glyph’s outline because of the big difference in the extents along the three axes.
A similar approach to my work is presented by Sigg et al in [2]. They also provide a library for OpenGL that is capable of drawing spheres, ellipsoids and cylinders. But in difference to my approach they use point sprites likes [5] and [10]. They also include an additional transformation step: the quadric is at the beginning in parameter space, where the quadric’s determining matrix is a diagonal, normalized one. Then it is transformed via a so-called variance matrix into the known object space. The calculation of the ray-quadric intersection is done in this parameter space, and they also include advanced techniques like deferred shading, soft shadows and silhouette lines.

Another interesting ray-casting system was developed by Pabst et al. [4]. The main
difference of their work is the use of NURBS surfaces instead of implicit quadrics. In a preprocessing stage every surface patch is subdivided into sub-patches and the convex hull of the sub-patches’ control points is generated. From this convex hulls a bounding volume consisting of many triangles is built for each surface to achieve a good approximation to the surface itself. For every fragment that is rasterized due to rendering the bounding volume the intersection point and normal is computed via a Newton Iteration. Also NURBS trimming curves are supported to generate formidable effects like cutting a logo out of a patch. Quadrics could also be composed of several NURBS surfaces making this technique very versatile, but also not as efficient as tailored shaders for specific implicit quadrics.

Figure 2.7.: Screenshot from [4] (left: NURBS scene, right: together with 9 NURBS trimming curves)

For visualizing macro-molecular structures, a prototype application has been developed by Offen and Fellner ([7]). They developed a core application responsible for the structure of a molecule and a plugin system, where visualization styles can be integrated as plugins. The main visualization styles provided are: Ball and Stick, Sticks, Spacefill, Ribbons and Surface. For the first 3 styles mainly spheres and cylinders are used which are rendered in a similar fashion like in my work through the help of special vertex and fragment shaders (GPU-based ray-casting). Fallback solutions are also provided if the hardware is not able to use custom shaders. Ribbons are rendered as Catmull and Clark subdivision surfaces and the surfaces are created with an algorithm based on reduced surfaces combined with adaptive tessellation resulting in triangular and quadrangular Bézier patches. In the BioBrowser an arbitrary selection of atoms, chains etc. can be taken which then can be annotated with metadata. These annotations are stored in a database and can be accessed via a SOAP web service or locally through an ODBC interface.
For rendering generalized cylinders Stoll et al. use a splatting technique that is also similar to this work: they render a quad on an segment’s approximating silhouette of the generalized cylinder and then interpolate normals and depth values ([12]).
3. Overall Approach

3.1. Overview

The common way of drawing a primitive is by dividing it into many triangles which are then passed to the OpenGL pipeline where they are rendered. The main problems with this approach are:

- for very near objects you need many triangles to get good visual performance
- for distant objects some triangles collapse to one single pixel

This leads to high transfer rates slowing down the whole rendering. The storage costs for scenes consisting of masses of triangles are also very high. There are some remedies to partially solve these problems, but avoiding them seems more appropriate for me.

Another possibility is raytracing: Every object is stored very compact (for a sphere only the center and the radius are needed) and the resulting image has a very high quality. This is achieved by sending rays through the pixels of the screen; for every ray the intersections with the objects in the scene are calculated and even refracted/reflected rays are evaluated. The problem with this method is that the whole calculation is done on the CPU resulting in low frame rates.

This work follows a similar approach. A raytracing is done, but without reflection/refraction and on the GPU. To let the GPU do the computation, a shader program consisting of a vertex and a fragment shader is installed. A vertex shader in general replaces the part of the fixed functionality pipeline, that processes every vertex without knowledge of the primitive type (basically transformation into clip space and preparation of values for the fragment shader). The special vertex shader used computes an approximation of the object’s silhouette. This can be done through different types of 2D-objects. It is possible to take just one single big point, which delivers good approximations for round silhouettes like that of a sphere, but due to its equal width and height it is unsuitable for outlines with different extents in the x- and y-axis. It is also not possible to let some values be incrementally calculated because you have just one run through one vertex unit without interpolation of values. But taking a polygon on the other side has the drawback, that many vertices have to be sent to the GPU and processed. The transfer can be improved by display lists and maybe by the use of geometry shaders.

After the vertex shader is finished, the primitives are assembled and rasterized. For every pixel emerging from this last step a fragment shader is applied that normally does the lighting calculation and texturing. In this work the special fragment shader does the intersection computation. It has to be mentioned that the per pixel calculation is much faster when tesselating the objects (no ray-object intersection has to be computed). So as a rule of thumb it can be said that the more pixels result on the screen, the faster it is with tesselation in comparison to ray-casting.

An overview of how the rendering process works is shown in Figure 3.1:

- four vertices are sent to the vertex shader that also gets some additional input:
– some attributes individual to the object (radius, height etc.)
– optionally an attribute representing a translation of the object can be given (via
  a special shader)
– the OpenGL state is given implicitly

• the vertex shader calculates for every vertex one corner of the object’s silhouette in
  a predefined order
• it also gives the positions of the corners in local coordinates to the fragment shader
  through varying variables
• this so-called splat is then rasterized by the fixed OpenGL pipeline between vertex
  and fragment shader (amongst other activities)
• in the fragment shader for every fragment that emerged from the rasterization a
  ray-intersection with the object happens, resulting in a intersection point in eye
  coordinates together with a normal and a depth value or the fragment is discarded
  when no intersection appears
• these values are then used to perform the phong lighting calculation for this fragment

![OpenGL pipeline diagram](image)

Figure 3.1.: OpenGL pipeline

An advanced version of the rendering process is presented in Figure 3.2. When a scene
consists of many objects, you can easily find viewpoints from where several objects lay
one after another. Then the expensive lighting calculation is done several times per pixel,
but this is superfluous, because only one pixel will be visible at the end. To greatly
improve the performance you can defer the lighting to a second step where it is done only
once per pixel. I.e. the position and normal in eye coordinates are passed to a deferred
fragment (illumination) shader while the object type’s fragment (intersection) shader only
computes the intersection. This deferred shading is also applied by e.g. [2]. One major
disadvantage with this technique is that because the illumination is done in a second step it is complicated to draw every object with different colors, material properties or textures.

3.2. General Properties of the Technique

3.2.1. Usage

Before a primitive can be rendered with the framework, it has to be initialised. In this step the shaders are loaded from file, compiled and then linked together to shader programs. Afterwards the shaders can be used to render primitives. The normal way of using is shown in Listing 1:

```java
for (Object o in objects) do
{
    activateShader(o.getType());
    render(o);
    deactivateShader(o.getType());
}
```

Rendering one object means sending all parameters to the shader program that are needed to know (e.g. radius, position) and then calling a previously compiled display list that actually contains the four vertices forming the silhouette.

The activation and deactivation of shader programs costs much time, so an optimization is straightforward:

![Advanced OpenGL pipeline](image)

**Figure 3.2.:** Advanced OpenGL pipeline
for (ObjectType t in objectTypes) do
{
    activateShader(t);
    for (Object o in t.objects) do
    {
        render(o);
    }
    deactivateShader(t);
}

For convenience both methods are supported in the framework (see section 5.3).

3.2.2. Integration into OpenGL

One important goal of this work is a seamless integration in OpenGL, which is achieved to a great extent. It is possible to render normal triangular meshes mixed with primitives from the framework. When rendering a primitive, the current OpenGL state is queried for many purposes:

- the whole computation of the object is done in object coordinates, which is then transformed into screen coordinates via the current model-view-projection matrix → projective transformations are transparently possible (e.g. arbitrary ellipsoids can be rendered by scaling a sphere appropriately)
- the near-clipping-plane is needed to perform an early z-test
- material properties as well as the light source properties / states are used for the lighting calculation
- user clipping features can be used (see 3.4.3)
- backface culling could be used, but is not implemented
- texturing is possible with limitations: only one texture can be used to texture all the primitives (normal OpenGL rendering is possible with multiple textures); however, it is not integrated
- display lists can also be used to encapsulate rendering potentially many primitives inside one list

The OpenGL requirements for the framework to be able to operate are:

- *OpenGL 2.0* or higher (for the use of shaders)
- the extension *GL_ARB_TEXTURE_FLOAT* (for floating point general purpose textures)
- the extension *GL_EXT_FRAMEBUFFER_OBJECT* (for deferred shading)
3.3. Vertex Shaders

3.3.1. Silhouette

The main task of the vertex shader is to determine the silhouette of the primitive being rendered. This work uses an approximation based on a quad which is quite flexible and offers a fairly good approximations of the objects’ outlines. The calculation is done in object coordinates which simplifies the calculation heavily because only the standard cases, i.e. sphere, cylinder and cone, of the implicit representations have to be considered, but the complex cases, e.g. ellipsoid, are also possible as a result. After that the quad is transformed into clip space (through the model-view-projection matrix) and sent to the rasterization unit. The coordinates in object space are also interpolated and given to the fragment shader, because it uses them for the ray-primitive intersection.

For all three primitive types considered so far (sphere, cylinder and cone) there is a special case: when the camera is inside the object, it covers the whole screen. The reason for this is that the silhouette only depends on the position of the camera when it is outside the object; but when inside it also depends on the viewing vector and thus is very difficult to determine. To make it even harder special cases exist in which the silhouette has holes in it, which can not be represented by a quad. As an example consider a camera inside a cylinder that is looking straight to the top as shown in Figure 3.3. To determine if the eye point $E$ is inside the primitive, $E$ must get inserted into the implicit representation of the primitive typ, the resulting value has to be lesser than 0 and also any additional constraints (like $0 \leq y \leq h$ at cylinder and cone) must be fulfilled.

![Figure 3.3.: View from inside the cylinder upwards](image)

Important is the order of the silhouette’s vertices, which must be anti-clockwise. Otherwise it would be discarded if backface culling is activated because it is rear-facing. After the silhouette was computed, the vector to the current corner has to be calculated in local space, because it is needed for the intersection (see 3.4.1. The eye point is also translated by the negative center (after determination of the silhouette), because the fragment shader assumes that the primitive is at the origin, which can be seen as a simple translation. If the eye point is also translated, the normal can also be determined in this reduced case, but in a simpler way. This works because for the normal the actual position does not matter,
only the relative position on the surface counts. The local position of the quad’s corners, however, is not translated, because it is needed to compute the real point in local space, which is then transformed to normalised device space.

3.3.2. Acceleration

Between a vertex and a fragment shader varying variables can be defined that are written in the vertex shader and then interpolated perspective-correct across the primitive’s quad. These interpolated values can then be read from the fragment shader, allowing the vertex shader to precompute some values for the fragment shader resulting in better performance because the vertex shader is executed four times per quad, while the fragment shader is executed for every pixel emerging from the rasterization, i.e. for the whole screen in the worst case. This interpolation can be used for two different kinds of acceleration:

- an incremental ray tracing can be used by precomputing some values needed for the ray-primitive intersection speeding up the intersection shader
- after calculating the intersection point and normal in parameter space in the intersection shader they have to be transformed to eye space which is done via matrix-vector multiplications; these multiplications can be shortened to vector-scalar multiplications by doing the matrix-vector multiplications in the vertex shader

I want to explain why the second kind of interpolation is correct. Interpolation among a primitive can be seen as a function $f$:

$$f(x_1, x_2, x_3, x_4) = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + x_4 \cdot w_4 = x,$$

where $x_i$ are vectors (e.g. position, normal) and $w_i$ are scalar values representing the weights of the quad’s four corners. The transformation is just a multiplication:

$$M \cdot x = M \cdot x_1 \cdot w_1 + M \cdot x_2 \cdot w_2 + M \cdot x_3 \cdot w_3 + M \cdot x_4 \cdot w_4,$$

where $M$ is a homogeneous 4x4-matrix. The interpolation forms a homomorphism:

$$f(M \cdot x_1, M \cdot x_2, M \cdot x_3, M \cdot x_4) = M \cdot x_1 \cdot w_1 + M \cdot x_2 \cdot w_2 + M \cdot x_3 \cdot w_3 + M \cdot x_4 \cdot w_4 = M \cdot x = M \cdot (x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + x_4 \cdot w_4),$$

i.e., it makes no difference if the interpolated values are transformed (transformation in the fragment shader) or the transformed values are interpolated (transformation in the vertex shader).

Another acceleration is to give the center of the primitive as an attribute to the shader. Then the silhouette determination becomes a little bit trickier, but the intersection shader can stay the same. The advantage is that you do not have to manually translate the primitive (by calls to ‘glTranslate(x, y, z)’), but just give the center to the shader which results in a great performance gain. This center is given in local space, so the translation is done after the translation via the inverse model-view matrix.

3.4. Fragment Shaders

3.4.1. Intersection Shaders

The calculation of the intersection point and normal is done in object coordinates, i.e. the center of the primitive lies at the origin, no matter if the primitive is translated by the
current model-view-matrix or by a center given to the vertex shader or both. The vertex shader has the responsibility to give the parameters needed in the right format to the intersection shader. Therefore the calculation works like normal ray tracing on the CPU: From the camera a ray is sent through every pixel of the silhouette after rasterization (see Figure 3.4). Then the intersection point between the ray and the primitive is determined and the normal is derived from it. This is only possible, if the object is geometrically describable, which can be very hard for general objects (e.g. a car). The depth value written in the depth buffer is the z-coordinate of the intersection point in normalised device space. The silhouette itself is not shown, since it is just an implement for the construction of the rays.

The intersection point and normal are transformed into eye space by applying the model-view and normal matrix, respectively. This transformation is done in the vertex shader as mentioned in 3.3.2, so only vector-scalar multiplications are needed. This is possible because a ray is described by the equation \( P = E + \lambda \cdot \vec{s} \), where \( E \) is the eyepoint and \( \vec{s} \) the ray-vector (vector from the eye point to the fragment).

Instead of transforming the resulting point \( P, E \) and \( \vec{s} \) are transformed in the vertex shader and then given to the fragment shader. Thus some varying variables have to be defined for this purpose. The depth value is obtained from the point in normalised device space, so the transformation is done by the model-view-projection matrix, of course also in the vertex shader. The \( w \)-clip has to be done in the fragment shader because it is no homomorphism:

\[
f(z_1, z_2, z_3, z_4) = z_1 \cdot a_1 + z_2 \cdot a_2 + z_3 \cdot a_3 + z_4 \cdot a_4 = z \\
f(w_1, w_2, w_3, w_4) = w_1 \cdot a_1 + w_2 \cdot a_2 + w_3 \cdot a_3 + w_4 \cdot a_4 = w,
\]

where \( a_i \) are the weights of the vertices; \( z_i \) and \( w_i \) are scalar values standing for the \( z \)- and \( w \)-coordinates of the vertices’ positions in clip space, respectively. But dividing the interpolated values is different from interpolating the quotient:

\[
\frac{z}{w} = \frac{z_1 \cdot a_1 + z_2 \cdot a_2 + z_3 \cdot a_3 + z_4 \cdot a_4}{w_1 \cdot a_1 + w_2 \cdot a_2 + w_3 \cdot a_3 + w_4 \cdot a_4} \neq \frac{z_1}{w_1} \cdot a_1 + \frac{z_2}{w_2} \cdot a_2 + \frac{z_3}{w_3} \cdot a_3 + \frac{z_4}{w_4} \cdot a_4 = f\left(\frac{z_1}{w_1}, \frac{z_2}{w_2}, \frac{z_3}{w_3}, \frac{z_4}{w_4}\right).
\]
The normalisation of the depth value (it has to be in the interval $[0, 1]$) on the other side can also be prepared in the vertex shader. Two vectors of dimension 2 are needed to accomplish the task of computing the normalised depth value. The real depth value is:

$$z = \left( \frac{P_{cz}}{P_{cw}} + 1 \right) \cdot 0.5 = \frac{P_{cz} + P_{cw}}{2 \cdot P_{cw}} \tag{3.1}$$

, where $P_c$ is the point $P$ in clip space. Because, as mentioned before, $P = E + \lambda \cdot \vec{s}$ and the transformed point is $P_c = M \cdot (E + \lambda \cdot \vec{s}) = M \cdot E + \lambda \cdot M \cdot \vec{s}$, where $M$ is the model-view-projection matrix. So to obtain the z- and w-coordinates, two intermediate values are introduced:

$$ep_p = M \cdot E$$
$$rv_p = M \cdot \vec{s}$$

In the first vector $ep_{pn} = (ep_{pz} + ep_{pw}, 2 \cdot ep_{pw})^T$ is written and in the second one $rv_{pn} = (rv_{pz} + rv_{pw}, 2 \cdot rv_{pw})^T$; in each vector in the first component the z-component plus the w-component is written, which stands for the upper part of 3.1. The second component is multiplied by 2, representing the lower part of 3.1. The x- and y-coordinates of $P_c$ are of no interest. Now only the normalised point in clip space has to be determined:

$$P_{cn} = ep_{pn} + \lambda \cdot rv_{pn};$$

and the depth value becomes to

$$z = \frac{P_{cn_z}}{P_{cn_w}}$$

At last the calculated values are stored in two buffers to let the deferred shader have access to it. I will come to this again in 3.4.3.

A problem arising from the ray-tracing is the z-clip. Consider the following situation: The camera is looking onto a sphere behind a cylinder. The first intersection (red point) with the cylinder lies behind the near-clipping-plane, so it is clipped away. The second intersection (green point) is revoked by the cylinder intersection shader, because the first intersection is in front of the second one. So what you see is the sphere (purple point), instead of the cylinder (see Figure 3.5). To overcome this problem, a manual z-clip has to be performed. The first intersection is tested against the near-clipping-plane and if necessary discarded in which case the second intersection is taken. Then the cylinder would be visible in the previous example instead of the sphere, which is correct. Because the z-near clipping plane is specified in normalised device space, this comparison is done with the depth value after the w-clip.
To be able to do compute the position and normal in eye space, three vectors of dimension 3 need to be interpolated from the four corners of the quad for every fragment: \( rv_e = M \cdot \vec{s} \) for the position computation, where \( M \) is the current model-view matrix and \( \vec{s} \) the vector from the eye point to the corner (which is then interpolated across the quad resulting in the ray-vector); \( ep_n = N \cdot E \) and \( rv_n = N \cdot \vec{s} \) are needed for the normal calculation, where \( N \) is the normal matrix. The real position in eye space then is: \( P = M \cdot (E + \lambda \cdot \vec{s}) = 0 + rv_e = rv_e \). The 0 in the third term is the eye point in eye space, which is by definition the origin. For the normal no universal formula can be given, it has to be derived from the position \( P \) by applying the gradient operator \( \nabla \) to the object’s implicite representation.

### 3.4.2. Illumination Shader

As illumination shader a phong shader is chosen simulating the behaviour of the conventional OpenGL lighting on a per-pixel basis. The shader only needs to know the position and normal of the fragment in eye space currently rendering together with the current OpenGL state to perform lighting. To simulate OpenGL behaviour as good as possible only the enabled lights are involved in rendering. But this part of the state is not given implicitly to a shader, so an array of length maximum number of lights, which contains for every light source the information enabled or disabled, is given to it.

### 3.4.3. Deferred Shading

**Values & Conventions**  
The deferred shader is the second phase of the rendering process. It consists of a shader program, which itself is created from a vertex and a fragment shader object, and an own framebuffer with two textures and a depth buffer. Before starting to draw the scene, the framebuffer is bound and it is specified to write in two buffers, i.e. the two textures. In the fragment shader the depth value is written to the normal variable `gl_FragDepth` to enable automatic z-clip. The first texture contains the diffuse material color (only the rgb part) and the z-coordinate of the pixel’s position in clip space in that order. The second one contains the normal in eye coordinates and the w-coordinate.
of the pixel’s position in clip space. Actually not the z- and w-coordinates themself are stored in the textures, but \((z + 2 \cdot w)\) for z and \((2 \cdot w)\) for w. The reason for this is that in the intersection shaders the coordinates are available in exactly this way (see 3.1) and the intersection shaders are more time-critical than the deferred shader, since they are executed more often. Extracting the real values is just simple subtraction and division. I will explain in one moment, why these values are needed.

After the whole scene has been drawn, the framework switches back to the normal framebuffer and renders the scene by sending a quad covering the whole screen to the vertex shader. This quad is not transformed but taken as position just as it is. Additionally the first texture coordinate is specified (it ranges from \((0 \ 0)\) in the left bottom corner to \((1 \ 1)\) in the right top corner) and the x- and y-coordinates of the position in normalised device space are given to the fragment shader as varying variables. Both textures have to be assigned to uniform samplers from which the fragment shader reads the necessary information, taking the interpolated texture coordinate. The texture filter has to be nearest, because it would be a fatal error to interpolate the normals, positions or diffuse colors, because neighboring pixels need not necessarily belong to the same primitive.

Illumination  Mixing tesselated objects rendered by OpenGL’s fixed functionality pipeline with primitives rendered by the shader library leads to one problem in the deferred shader: It does not know if a fragment was rendered by the one way or the other. But he has to distinguish it because only in the latter case a lighting calculation has to be done. If the pixel stems from fixed functionality, the pixel’s color is written into both buffers represented by the two textures. Because every single value of the color is clamped to \([0, 1]\), the shaders increase the blue value (third component) of the diffuse color by 2 before writing it into the texture. The deferred shader now only has to check if the third value of the first texture is greater than 1. If this is the case, the lighting calculation has to be performed for this fragment (first the blue value is decreased by 2 to get the real value). Otherwise the fragment was calculated through fixed functionality pipeline and the content of the first texture is taken as fragment color; it does not matter which texture is taken in that case, because the framebuffer is specified to write in both buffers.

To give the illumination shader the pixel’s position in eye space it first has to be computed. The z- and w-coordinates of the position in clip space is know from the textures. The x- and y-coordinates are also known, because the fragment’s position in normalised device space comes from the vertex shader, and by reversing the w-clip (multiplying the x-/ y-coordinate by the w-coordinate) the clip coordinate is given. By assuming that the projection matrix does not change between the objects in the scene the clip space can be seen as view- and position-independent which holds not for the eye space because it depends on the current model-view matrix which normally changes between drawing of two objects. The position is now transformed back to eye space with the help of the inverse projection matrix and the illumination shader is called with the position, normal and diffuse material color.

Limitations of / Problems with deferred shading  The reason why the diffuse material color is passed lies in the nature of the two-phase rendering process. In the first phase the OpenGL state, including material properties, can change repeatedly. If the phong shading would be immediately, i.e. no deferred shading used, the current state is available. But in the second phase only the final state, after all state changes took place, can be seen.
To make the whole state available, every fragment would have to write many properties, e.g. material color, near- and far-clipping plane, into several textures resulting in a slower shading. So only the diffuse material color is chosen, because it has a great influence on the resulting color of the fragment and some properties like light source positions and colors just should not change between two objects, because it would irritate the viewer (unless this is a desired feature).

Texturing is theoretically possible, but not implemented, and again with the problem of different states, only one texture would be taken for the whole scene.

**User Clipping** User clipping is also done in this phase. Normally the stages between the vertex and the fragment shader care about it, but in the vertex shader the real geometry is not known, so it can only be done after ray-casting. In the implicit state no information is given about the enabled state of the clipping planes, as it is with the light source state (see 3.4.2). Therefore an array of length maximum number of clip planes is transferred to the shader. As the planes are stored in eye coordinates, the fragment in eye space has to be tested against the plane. This is done by a dot product; if it is negative, the fragment is out with respect to that clipping plane and thus discarded. Of course, only enabled clipping planes are tested. However, this offers not a full support of user clipping, because it could be possible, that the first primitive along a ray is clipped away by a specified clip plane, but the next primitive, which is instead behind the clipping plane, should be visible (see Figure 3.6). But it is not visible because the deferred shader does not know which object lies behind the fragment. To overcome this problem, user clipping had to be performed in the intersection shaders. Then the shader of the first primitive would discard its fragments, but the other one’s primitive would still be visible. Because this would result in much slower intersection shaders, I decided against this full support of user clipping. This can result in strange shadows like in Figure 3.7.
Figure 3.6.: User Clipping Problem

Figure 3.7.: User Clipping Screenshots (left: full support, right: partly support)
4. Special Shaders

This chapter contains detailed descriptions of the shaders for the three primitive types (sphere, cylinder, cone) covered in this work. Every section is divided into a subsection for the silhouette determination and one for the intersection issue.

4.1. Sphere

Implicit representation: \( x^2 + y^2 + z^2 = r^2 \), where \( r \) is the radius of the sphere.

4.1.1. Vertex Shader

The silhouette \( S \) of an object is the set of points \( p \): \( S = \{ p \in O \mid n(p) \perp \overrightarrow{pE} \} \), where \( O \) is the set of points defining the object, \( E \) the eye point and \( n(p) \) the normal at the point \( p \), i.e. the surface normal has to be orthogonal to the vector pointing to the eye point. In Figure 4.1 a profile of a sphere is shown. An explanation of the labels follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>position of the eye point (camera)</td>
</tr>
<tr>
<td>( C )</td>
<td>center of the sphere</td>
</tr>
<tr>
<td>( g )</td>
<td>radius of the sphere</td>
</tr>
<tr>
<td>( M )</td>
<td>center of the silhouette circle</td>
</tr>
<tr>
<td>( r )</td>
<td>radius of the silhouette circle</td>
</tr>
</tbody>
</table>

*Table 4.1.*: Legend Sphere Silhouette
In the case of a sphere the silhouette is a circle on the plane that is orthogonal to $\overrightarrow{CE}$ and goes through $M$. Three rectangled triangles form the basis for the computation:

\begin{align*}
e^2 &= g^2 + s^2 \quad (4.1) \\
s^2 &= r^2 + b^2 = r^2 + (e - m)^2 = r^2 + e^2 + m^2 - 2 \cdot e \cdot m \quad (4.2) \\
g^2 &= r^2 + m^2 \quad (4.3)
\end{align*}

By inserting 4.2 and 4.3 in 4.1, one gets:

\[ r^2 = e \cdot m - m^2 \quad (4.4) \]

Now inserting 4.4 in 4.3 yields:

\[ m = \frac{g^2}{e} \quad (4.5) \]

$M$ can be read off directly:

\[ M = C + \frac{g^2}{e^2} \overrightarrow{CE} \quad (4.6) \]
Only $r$ is missing, which can be obtained by inserting $4.5$ in $4.3$:

$$r = g\sqrt{1 - \frac{g^2}{e^2}} \quad (4.7)$$

Because a quad is needed an arbitrary square containing the circle has to be specified. Therefore an up-vector $\vec{u}$ and a right-vector $\vec{r}$ are defined in a way that $\vec{u}$, $\vec{r}$ and $\overrightarrow{CE}$ are orthogonal to each other: $\vec{r}$ is defined as $\overrightarrow{CE} \times \vec{y}$, where $\vec{y} = (0, 1, 0)^T$ defines the $y$-axis in object coordinates; if $\overrightarrow{CE} = (0, e, 0)^T$, then $\vec{r} = (0, 0, 1)^T$. With $\vec{r}$ and $\overrightarrow{CE}$ known, $\vec{u} = \overrightarrow{CE} \times \vec{r}$.

After that $\vec{u}$ and $\vec{r}$ are scaled to have the length $r$. Then the four corners of the quad become: $M \pm \vec{u} \pm \vec{r}$.

### 4.1.2. Intersection Shader

For the fragment shader two variants are possible: An explicit and an incremental one. The incremental variant as opposed to the explicit one needs two values that are being interpolated across the quad. With these additional two values the calculation is shorter and thus faster. The problem is that it only works if the eye point is outside the sphere. So the shader can differentiate whether the eye point is inside or not and then use the appropriate variant, i.e. explicit when inside and incremental when outside. But this conditional slows down the shader (see section 6.3). Thus the framework can be initialized to either load the explicit shaders or the implicit ones, which are faster, but are not able to render a sphere when the viewer is inside it.

**a. Explicit Solution**

The explicit variant is the easier one. Only the explicit representations (as mentioned before in object space) of the ray and the sphere are needed as shown in Figure 4.2.
The ray is just a straight line defined by:

$$E + \lambda (S - E) =: E + \lambda \cdot \vec{s}, \; \lambda \in \mathbb{R}$$  \hspace{2cm} (4.8)

And the sphere is given by the set of points $R$ that fulfill the following equation:

$$R_x^2 + R_y^2 + R_z^2 = g^2$$  \hspace{2cm} (4.9)

Because the intersection between the sphere and the ray needs to be determined, this intersection point has to lie in both of them, i.e. the same point $P$ has to conform to both equations, and so inserting (4.8) in (4.9) leads to:

$$(E_x + \lambda s_x)^2 + (E_y + \lambda s_y)^2 + (E_z + \lambda s_z)^2 = g^2$$  \hspace{2cm} (4.10)

Which simplifies by the definitions:

$$s := s_x^2 + s_y^2 + s_z^2$$
$$p := E_x \cdot s_x + E_y \cdot s_y + E_z \cdot s_z$$
$$q := E_x^2 + E_y^2 + E_z^2 - g^2$$

to

$$\lambda^2 \cdot s + 2 \cdot \lambda \cdot p + q = 0$$  \hspace{2cm} (4.11)
\( q \) is also the value of the eye point inserted into the implicit representation, so it can be used to differentiate the two variants. The solutions for 4.11 are:

\[
\lambda_{1,2} = -\frac{p}{s} \pm \sqrt{\frac{p^2}{s^2} - \frac{q}{s}} \tag{4.12}
\]

where \( \lambda_1 \) belongs to the upper term, i.e. the one with the minus, and therefore \( \lambda_2 \) belongs to the lower term. The intersections \( P_{1,2} \) are now obviously: \( P_{1,2} = E + \lambda_{1,2} \cdot s \). The first intersection \( P_1 \) is normally the right one since it is occluded by the second one. But if \( \lambda_1 < 0 \), \( P_2 \) has to be taken because \( P_1 \) lies behind the viewer. \( P_2 \) also has to be taken if the manual z-clip (see Chapter 3.4.1) discards \( P_1 \).

Two interpolated 3-dimensional vectors representing \( s \) and \( E \) are needed as input, in addition to the value \( q \) that is constant for the whole sphere, since it only depends on the camera position \( E \) and the radius of the sphere \( g \).

b. Incremental Solution

The incremental method was developed by Gumhold in [3]. Here I assume a rotation of the axes such that the x-axis is along the silhouette (analogous \( \vec{r} \) in the vertex shader) and the z-axis is the viewing vector. This corresponds to the vectors \( \vec{x}' \) and \( \vec{z}' \) in Figure 4.2. First of an auxiliary calculation is necessary:

\[
e^2 \cdot (4.7)^2 \rightarrow e^2 \cdot r^2 = g^2 \cdot (e^2 - g^2) \rightarrow e^2 - g^2 = \frac{e^2 \cdot r^2}{g^2} \tag{4.13}
\]

Then inserting 4.13 in 4.1 leads to:

\[
s^2 = \frac{e^2 \cdot r^2}{g^2} \tag{4.14}
\]

With 4.2 transformed and then substituted for \( r^2 \) one gets:

\[
s^2 = \frac{e^2}{g^2} \cdot (s^2 - b^2) \tag{4.15}
\]

This is equal to

\[
e^2 \cdot b^2 = s^2 \cdot (e^2 - g^2) \tag{4.16}
\]

Now 4.1 is solved for \( s^2 \) and inserted in 4.16:

\[
e^2 \cdot b^2 = s^4 \tag{4.17}
\]

Which leads to

\[
e \cdot b = s^2 \tag{4.18}
\]

With definition of \( \vec{v} = \vec{ES} := (x, y, b)^T \) and \( \vec{e} = \vec{0E} = (0, 0, -e)^T \triangleq E \), 4.18 leads to:

\[
\vec{e} \cdot \vec{v} = -e \cdot b = -s^2 \tag{4.19}
\]

Replacing \( s \) by \( \vec{v} \) in 4.10 leads to:

\[
g^2 = E_x^2 + 2 \cdot \lambda \cdot E_x \cdot v_x + \lambda^2 \cdot v_x^2 + E_y^2 + 2 \cdot \lambda \cdot E_y \cdot v_y + \lambda^2 \cdot v_y^2 + E_z^2 + 2 \cdot \lambda \cdot E_z \cdot v_z + \lambda^2 \cdot v_z^2
\]

\[
= E_x^2 + E_y^2 + E_z^2 + 2 \cdot \lambda \cdot (E_x \cdot v_x + E_y \cdot v_y + E_z \cdot v_z) + \lambda^2 \cdot (v_x^2 + v_y^2 + v_z^2)
\]
Which shortens with $|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2} =: v$ and $|\vec{e}| = \sqrt{e_x^2 + e_y^2 + e_z^2} = \sqrt{e^2} = e$ to:

$$g^2 = e^2 + 2 \cdot \lambda \cdot (E_x \cdot v_x + E_y \cdot v_y + E_z \cdot v_z) + \lambda^2 \cdot v^2$$

Since $\vec{e} \equiv E$ and with the help of 4.19, this is equal to:

$$g^2 = e^2 - 2 \cdot \lambda \cdot s^2 + \lambda^2 \cdot v^2 \quad (4.20)$$

Applying 4.1 $g^2$ to this results in:

$$0 = s^2 - 2 \cdot \lambda \cdot s^2 + \lambda^2 \cdot v^2 \quad (4.21)$$

Now I divide by $s^2$:

$$0 = 1 - 2 \cdot \lambda + \frac{v^2}{s^2} \cdot \lambda^2 \quad (4.22)$$

Since $v$ was defined as $|\vec{v}|$, $\vec{v}$ can be inserted:

$$0 = 1 - 2 \cdot \lambda + \frac{x^2 + y^2 + b^2}{s^2} \cdot \lambda^2 \quad (4.23)$$

In 4.14 $s^2$ is defined, leading to:

$$0 = 1 - 2 \cdot \lambda + \frac{x^2 + y^2 + b^2}{e^2 \cdot r^2} \cdot g^2 \cdot \lambda^2$$

$$= 1 - 2 \cdot \lambda + \left( \frac{b^2 \cdot g^2}{e^2 \cdot r^2} + \frac{x^2 + y^2}{r^2} \cdot \frac{g^2}{e^2} \right) \cdot \lambda^2 \quad (4.24)$$

4.17 can be transformed to $b^2 = \frac{s^4}{e^2}$ and with 4.14 replaced for $s^2$:

$$b^2 = \frac{e^2 \cdot r^4}{g^4}$$

Now 4.24 can be written as:

$$0 = 1 - 2 \cdot \lambda + \left( \frac{r^2}{g^2} + \frac{x^2 + y^2}{r^2} \cdot \frac{g^2}{e^2} \right) \cdot \lambda^2 \quad (4.25)$$

Next a vector $\vec{q}$, which represents the vector from the center of the silhouette circle to the fragment, but scaled so that $q_x, q_y \in [-1, 1]$, is defined:

$$\vec{q} := \frac{1}{r} \cdot \left( \begin{array}{c} x \\ y \end{array} \right), \quad q := |\vec{q}| = \frac{1}{r} \cdot \sqrt{x^2 + y^2} \quad (4.26)$$

With these definitions 4.25 becomes to:

$$0 = 1 - 2 \cdot \lambda + \left( \frac{r^2}{g^2} + q^2 \cdot \frac{g^2}{e^2} \right) \cdot \lambda^2 \quad (4.27)$$

4.7 can be used to rewrite it:

$$0 = 1 - 2 \cdot \lambda + \left( 1 + \frac{g^2}{e^2} \cdot (q^2 - 1) \right) \cdot \lambda^2 = 1 - 2 \cdot \lambda + \left( 1 - \frac{g^2}{e^2} \cdot (1 - q^2) \right) \cdot \lambda^2 \quad (4.28)$$

30
Because the incremental variant only works, when the viewer is outside the sphere, \( e > g \) must be true. Thus a variable \( u := \frac{g}{e} \) is defined, so that \( u \in [0, 1] \) (\( g \) and \( e \) are positive). Next I define \( a := \sqrt{1 - q^2} \) and \( b := u \cdot a \). This leads to \( b^2 = u^2 \cdot a^2 = \frac{g^2}{e^2} \cdot (1 - q^2) \). By inserting this in 4.27 I obtain:

\[
0 = 1 - 2 \cdot \lambda + (1 - b^2) \cdot \lambda^2
\]  

(4.29)

Solving this by the quadratic formula offers the following solutions:

\[
\lambda_{1,2} = \frac{1}{1 \pm b}
\]  

(4.30)

Only if \( q^2 = \frac{x^2+y^2}{r^2} \in [0, 1] \), the fragment is inside the silhouette circle, defined by \( r^2 = x^2 + y^2 \), and thus inside the sphere. This offers an early exit point for the shader: *If \( q^2 > 1 \), the fragment is discarded.* Because \( q^2 \leq 1 \) is true, if the shader proceeds, it can be derived, that also \( 1 - q^2 \in [0, 1] \), \( a \in [0, 1] \) and thus \( b \in [0, 1] \). But \( \lambda_2 = \frac{1}{1 - b} > 1 \), i.e. the intersection is behind \( S \) resulting in the backmost intersection. But the interesting one is the one in the front, which is

\[
\lambda = \lambda_1 = \frac{1}{1 + b} = \frac{1}{1 + u \cdot \sqrt{1 - q^2}}
\]  

(4.31)

Of course the manual z-clip can also decide to take \( \lambda_2 \) if necessary and the resulting point is \( P = E + \lambda \cdot \vec{s} \).

Less interpolated values are needed: Only 3 values (\( u \), \( x \) and \( y \)) can be combined in one vector, which is important because the GL shading language only copes with vectors of the dimension 4 on most platforms, and so three single floats would waste 3 vectors of dimension 4 resulting in less varyings left and worse performance. This vector also contains \( u \) because it has a constant value for the whole sphere and thus need not be calculated for every fragment anew. Since the four corners of the quad approximating the silhouette circle shape a square of length \( 2 \cdot r \), the local coordinates of the vertices can be taken directly as \( x \) and \( y \), because they were defined as \((\pm 1, \pm 1)^T\) which is equal to \( \vec{q} \) which defines a circle of radius \( r \).

c. Normal Computation

Applying the \( \nabla \)-operator to the implicit representation of a sphere, which is \( x^2 + y^2 + z^2 - g^2 = 0 \), yields:

\[
\vec{n} = \nabla \left( x^2 + y^2 + z^2 - g^2 \right) = (2 \cdot x, 2 \cdot y, 2 \cdot z)^T \triangleq 2 \cdot P
\]  

(4.32)

So the normal at a given point is just the point itself, because at the end the normal is normalised, so the constant factor 2 can be left out: \( \vec{n} = N \cdot P = N \cdot (E + \lambda \cdot \vec{s}) = ep_n + \lambda \cdot rv_n \) (see 3.4.1 for the explanation of the symbols). As mentioned in 3.4.3, normalisation is not necessary at this time.
4.2. Cylinder

Implicit representation: \( x^2 + z^2 = r^2 \) with the additional constraint \( 0 \leq y \leq h \), where \( r \) is the radius and \( h \) the height of the cylinder.

4.2.1. Vertex Shader

The calculation of the silhouette of a cylinder can be split into 2 different cases:

- Case 1 (Page 32): the viewer is inside the elongated cylinder, i.e. a cylinder with the same radius, but an infinite height
  - if the by the negative center translated eye point \( E \) inserted in the implicit formula is below or equal \( 0 \), i.e. \( E_x^2 + E_z^2 - r^2 \leq 0 \)

- Case 2 (Page 34): the camera is outside the elongated cylinder
  - if \( E_x^2 + E_z^2 - r^2 > 0 \)

a. Case 1 (inside)

The situation is depicted in Figure 4.3, where the dotted area shows the silhouette determined by the shader. An explanation of the labels are given in the table below it. As can be seen, the silhouette is just the upper edge circle. In OpenGL the camera is seen as a pinhole and thus all points on the cylinder under this upper circle are also covered by the rays through the filled circle’s pixels. When the camera is below the cylinder, the solution is analog, but this time with the lower edge circle.

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>position of the eye point (camera)</td>
</tr>
<tr>
<td>( C )</td>
<td>origin of the cylinder</td>
</tr>
<tr>
<td>( g )</td>
<td>radius of the cylinder</td>
</tr>
<tr>
<td>( h )</td>
<td>height of the cylinder</td>
</tr>
<tr>
<td>( M )</td>
<td>center of the silhouette circle</td>
</tr>
</tbody>
</table>

Table 4.2.: Legend Cylinder Silhouette (inside)
I define two variants for the two distinct places, i.e. variant 1 when above and variant 2 when below the cylinder. The center and radius of the silhouette circle can be obtained without calculation:

\[
M_{1,2} = \begin{pmatrix} C_x \\ C_y + h/2 \pm h/2 \\ C_z \end{pmatrix}
\]

\[r = g\]

And the corners for the quad are also obvious since the quad containing the circle can be rotated around the y-axis arbitrarily, so it does not matter how the quad is positioned, as long as it has the center \(M\) and the width \(2 \cdot r\):

\[
M \pm \begin{pmatrix} r \\ 0 \\ 0 \end{pmatrix} \pm \begin{pmatrix} 0 \\ 0 \\ r \end{pmatrix}
\]
b. Case 2 (outside)

This case is shown in Figure 4.4. The variables have the same meaning as in Case 1 (see Table 4.3). Basically the silhouette is the half of the surface, that is directed to the camera. But actually, the by this half generated quad has to be moved towards the viewer and its height / width must get adjusted.

![Figure 4.4.: Cylinder Silhouette (outside)](image)

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>position of the eye point (camera)</td>
</tr>
<tr>
<td>$E'$</td>
<td>position of the projected eye point</td>
</tr>
<tr>
<td>$C$</td>
<td>origin of the cylinder</td>
</tr>
<tr>
<td>$g$</td>
<td>radius of the cylinder</td>
</tr>
<tr>
<td>$h$</td>
<td>height of the cylinder</td>
</tr>
<tr>
<td>$M$</td>
<td>center of the silhouette circle</td>
</tr>
<tr>
<td>$y$</td>
<td>height of the silhouette circle</td>
</tr>
</tbody>
</table>

Table 4.3.: Legend Cylinder Silhouette (outside)

When you look at the cylinder from the top, i.e. the plane defined by the lower edge circle, it is just the same as with the sphere (see Figure 4.1). The reason, why it does not depend on the height of the eye point, is simple: The normals in the silhouette points are orthogonal to the view ray. The y-component of the normal is $\theta$ for any point on the
cylinder (see Page 39). So the dot product does not change if the height of the eye point changes, since this only affects the y-component of the viewing ray, which does not affect the dot product \(0 \cdot x = 0\). Now I can assume that the eye point lies in the same plane as the lower circle resulting in the point \(E' = (E_x, C_y, E_z)^T\). With this assumption the basic requirements for the calculation of the silhouette circle of the sphere are also met, and thus all results also hold here. The center and the width \(r\) of the silhouette quad become:

\[
M = C + \frac{g^2}{e^2} \overrightarrow{CE'} + \begin{pmatrix} 0 \\ \frac{h}{2} \\ 0 \end{pmatrix} \quad (4.33)
\]

\[
r = g \sqrt{1 - \frac{g^2}{e^2}} \quad (4.34)
\]

Naively \(h\) could be taken as height, but this is not correct as it disregards the real \(y\)-component of the eye point. As can be seen in Figure 4.5, not the whole cylinder is covered by a quad with the height \(h\), because the silhouette also includes the circles on the top and bottom, which is not covered by the formulas derived for the sphere silhouette, i.e. \(M\) and \(r\). In the figure the red lines delimit the visible space without height correction, the blue ones show it with height correction (the whole cylinder). The red dotted areas brand the parts of the cylinder that are discarded in the former case.

Now the correct height can be determined by computing the intersection from a ray.
through an otherwise not visible corner of the cylinder and the elongated quad. First the corner of interest \( T \) has to be determined, which depends on the current vertex and on the height of the eye point:

- for the two lower vertices a lower corner must be selected
  - if the eye point is below the center of the cylinder (\( E_y < C_y \)), the back side corner is taken: \( T = C - g \cdot \frac{\overrightarrow{CE}}{|\overrightarrow{CE}|} \)
  - otherwise (\( E_y \geq C_y \)), the front side corner is taken: \( T = C + g \cdot \frac{\overrightarrow{CE}}{|\overrightarrow{CE}|} \)
- the upper vertices only examines the upper corners
  - if the eye point is above the cylinder (\( E_y > C_y + h \)), the rear corner is the right one: \( T = C - g \cdot \frac{\overrightarrow{CE}}{|\overrightarrow{CE}|} + \begin{pmatrix} 0 \\ h \\ 0 \end{pmatrix} \)
  - otherwise (\( E_y \leq C_y + h \)), the front corner is right: \( T = C + g \cdot \frac{\overrightarrow{CE}}{|\overrightarrow{CE}|} + \begin{pmatrix} 0 \\ h \\ 0 \end{pmatrix} \)

With the right corner the intersection can be calculated. The ray is defined by \( E + t \cdot \overrightarrow{ET} \) and the intersection takes place exactly above or below the center of the silhouette quad \( M \). Thus the intersection point \( P \) is known partly: \( P = \begin{pmatrix} M_x \\ y \\ M_z \end{pmatrix} \). Since \( P \) must be part of the ray, \( P = E + t \cdot \overrightarrow{ET} \) must be solved for \( t \), leading to:

\[
  t = \frac{M_x + E_x}{ET_x}
\]

Now getting \( y \) is no problem: \( y = E_y + t \cdot ET_y \). A right vector \( \vec{r} \) is defined as \( \overrightarrow{CE} \times \vec{u} \) with \( \vec{u} = (0, 1, 0)^T \), leading to \( \vec{r} = \begin{pmatrix} -CE_x \\ 0 \\ CE_y \end{pmatrix} \). After that \( \vec{r} \) is scaled to have the length \( g \), which indeed is half of the width. Here the case that the eye point is directly above the center can not occur and thus \( \vec{r} \) can always be specified this way.

Finally the four corners of the quad can be computed:

\[
  \begin{pmatrix}
    M_x \pm r_x \\
    y \\
    M_z \pm r_z
  \end{pmatrix}
\]

### 4.2.2. Intersection Shader

Like with the sphere, two approaches are possible. An explicit variant can be used in any situation, while an incremental variant only works correct, if the camera is outside the elongated cylinder. A switch between these two variants is also possible, but again this comes along with performance losses. However, I decided to use the explicit version.
a. Explicit Solution

For this variant no additional interpolated values are needed. As already known from the sphere, the ray is just a straight line:

\[ E + \lambda \cdot \vec{s}, \; \lambda \in \mathbb{R} \]  \hspace{1cm} (4.35)

While the cylinder is defined by the set of points \( R \) that fulfill

\[ R_x^2 + R_z^2 = g^2, \; 0 \leq R_y \leq h \]  \hspace{1cm} (4.36)

with \( g \) and \( h \) as radius and height of the cylinder. The solution can be obtained again by inserting the ray equation into the cylinder equation:

\[ (E_x + \lambda s_x)^2 + (E_z + \lambda s_z)^2 = g^2 \]  \hspace{1cm} (4.37)

For convenience three intermediate values are defined:

\[ s := s_x^2 + s_z^2 \]
\[ p := E_x \cdot s_x + E_z \cdot s_z \]
\[ q := E_x^2 + E_z^2 - g^2 \]

\( q \) again holds the information, if the eye point is inside the elongated cylinder or not; but to check, if it is inside the cylinder, also the additional constraints must be fulfilled. With the values \( s \), \( p \) and \( q \), 4.37 becomes:

\[ \lambda^2 \cdot s + 2 \cdot \lambda \cdot p + q = 0 \]  \hspace{1cm} (4.38)

The solutions for this equation are:

\[ \lambda_{1,2} = \frac{-p}{s} \pm \sqrt{\frac{p^2}{s^2} - \frac{q}{s}} \]  \hspace{1cm} (4.39)

The intersection points \( P_{1,2} \) are given by \( E + \lambda_{1,2} \cdot \vec{s} \). But now the additional constraints must be checked. First of all, \( P_1 \) is tested, since it lies in front of \( P_2 \). So if the test \( 0 \leq P_1 \leq h \) is negative, \( P_2 \) is tested. If for \( P_2 \) the constraints do not hold, the fragment is discarded. This can happen, if the eye point is high above the cylinder, but relatively near, because than the lower hole of the cylinder is also visible; this scenario is shown in Figure 4.6, where the dotted gray lines depict the elongated cylinder, and the blue line shows the concrete ray, which produces no fragment at all. Addionally, the manual z-clip is performed (see Chapter 3.4.1), which can also be the reason for taking \( P_2 \) or discarding the fragment.

Two interpolated 3-dimensional vectors representing \( \vec{s} \) and \( E \) are needed as input, in addition to the value \( q \) that is constant for the whole cylinder.
b. Incremental Solution

As can be seen in the explicit variant, the calculation of $\lambda$ does not depend on the actual $y$-component of the eye point $E_y$. So the eye point can be transformed to $E' = (E_x, C_y, E_z)^T$ as in the outside case of the vertex shader. Now again the situation is the same as with the sphere (described in Figure 4.2) and thus the derivation can be inherited partly. The Equation 4.19 holds with the difference, that $\vec{v} = \vec{E}\vec{S} := (x, 0, b)^T = \vec{s}$. Inserting this into 4.37 results in:

$$g^2 = E_x^2 + E_z^2 + 2 \cdot \lambda \cdot (E_x \cdot v_x + E_z \cdot v_z) + \lambda^2 \cdot (v_x^2 + v_z^2)$$

The next step uses $|\vec{v}| = \sqrt{v_x^2 + v_z^2} =: v$, $|\bar{e}| = \sqrt{e_x^2} = e$ and $\bar{e} \triangleq E$ to come to

$$g^2 = e^2 - 2 \cdot \lambda \cdot s^2 + \lambda^2 \cdot v^2$$

(4.40)

Which is the same as in the sphere incremental version (4.20). Because of the different $\vec{v}$, 4.23 changes to:

$$0 = 1 - 2 \cdot \lambda + \frac{x^2 + b^2}{s^2} \cdot \lambda^2$$

(4.41)

Applying the same transformation steps, I end up with an equivalent equation for Equation 4.25:

$$0 = 1 - 2 \cdot \lambda + \left(\frac{r^2}{g^2} + \frac{x^2}{r^2} \cdot \frac{g^2}{e^2}\right) \cdot \lambda^2$$

(4.42)

Because of the missing summand $g^2$ in the numerator in the term inside the brackets, $\tilde{q}$ is defined differently:

$$\tilde{q} := \frac{1}{r} \cdot \begin{pmatrix} x \\ 0 \end{pmatrix}, q := |\tilde{q}| = \frac{1}{r} \cdot \sqrt{x^2}$$

(4.43)

Whereas equation 4.27 holds again. With the same definition for $u, a$ and $b$, this again can be transformed to

$$0 = 1 - 2 \cdot \lambda + (1 - b^2) \cdot \lambda^2$$

(4.44)
With the solutions:

\[ \lambda_{1,2} = \frac{1}{1 \pm b} \] (4.45)

An early exit test is not possible here, because the silhouette is not a circle approximated by a quad, but a rectangle, so the approximation is exact and no unnecessary fragments are rendered. However, there are unnecessary pixels, but this comes from the fact, that also the additional constraints \( 0 \leq y \leq h \) must be fulfilled. Again the interesting \( \lambda \) is the one in the front, which is

\[ \lambda = \lambda_1 = \frac{1}{1 + b} = \frac{1}{1 + u \cdot \sqrt{1 - q^2}} \] (4.46)

Of course the manual z-clip or the additional constraints can also decide to take \( \lambda_2 \) or discard the fragment if necessary. The resulting point clearly is \( P = E + \lambda \cdot \vec{s} \); attention should be paid to using the real eye point position \( E \) instead of the translated \( E' \).

Now only 2 interpolated values \((u, x)\) are needed, which again should be combined in one vector. Since the four corners of the quad defining the silhouette shape a square of length \( 2 \cdot r \), the x-component of the local coordinates of the vertices can be taken directly as \( x \), because the vertices were defined as \((\pm1, \pm1, 0)^T\).

c. Normal Computation

With the \( \nabla \)-operator applied to the implicit representation of a cylinder, which is \( x^2 + z^2 - g^2 = 0 \), one gets:

\[ \vec{n} = \nabla (x^2 + z^2 - g^2) = (2 \cdot x, 0, 2 \cdot z)^T \triangleq 2 \cdot \begin{pmatrix} P_x \\ 0 \\ P_z \end{pmatrix} \] (4.47)

Because in the deferred shader the normal is normalised, the constant factor 2 can be left out:

\[ \vec{n} = N \cdot \begin{pmatrix} P_x \\ 0 \\ P_z \end{pmatrix} = N \cdot \begin{pmatrix} E_x + \lambda \cdot s_x \\ 0 \\ E_z + \lambda \cdot s_z \end{pmatrix} = eP_n' + \lambda \cdot rv_n' \]

with

\[ eP_n' = N \cdot \begin{pmatrix} E_x \\ 0 \\ E_z \end{pmatrix} \]

and

\[ rv_n' = N \cdot \begin{pmatrix} s_x \\ 0 \\ s_z \end{pmatrix} \]

(N is the normal matrix). As mentioned in 3.4.3, normalisation is not necessary at this time.
4.3. Cone

Implicit representation: \( x^2 - (\frac{r}{h} \cdot y)^2 + z^2 = 0 \) with the additional constraint \( 0 \leq y \leq h \), where \( r \) is the radius at the top side and \( h \) the height of the cone. Normally it is written this way: \( x^2 + y^2 - (\frac{r}{h} \cdot z)^2 = 0 \), but I prefer the first variant, which causes the cone to ‘stand upright’ like the cylinder.

4.3.1. Vertex Shader

For the determination of a cone’s silhouette even more cases have to be distinguished:

- Case 1 (Page 40): the camera is inside the elongated cone, i.e. a cone, that is continued to both above and below, resulting in a cone with infinite height and radius, which have the same ratio like \( r \) to \( h \) – this is the case if the translated eye point is inside the cone without paying attention to the additional constraints: \( E_x^2 - (\frac{r}{h} \cdot E_y)^2 + E_z^2 \leq 0 \)

- Case 2 (Page 42): the camera is outside the elongated cone, but above it
  - if \( E_x^2 - (\frac{r}{h} \cdot E_y)^2 + E_z^2 > 0 \) and \( E_y \geq h \)

- Case 3 (Page 43): the camera is outside the elongated cone, but below it
  - if \( E_x^2 - (\frac{r}{h} \cdot E_y)^2 + E_z^2 > 0 \) and \( E_y \leq 0 \)

- Case 4 (Page 45): the camera is outside the elongated cone and in front of it (neither above nor below), but very low
  - if \( E_x^2 - (\frac{r}{h} \cdot E_y)^2 + E_z^2 > 0 \) and \( 0 \leq E_y \leq 0.1 \)

- Case 5 (Page 49): the camera is outside the elongated cone, in front of it
  - if \( E_x^2 - (\frac{r}{h} \cdot E_y)^2 + E_z^2 > 0 \) and \( 0.1 \leq E_y \leq h \)

a. Case 1 (inside)

This case is similar to the first case of the cylinder (see Page 32). In Figure 4.7 the situation is shown for two different camera positions (one above and one below the cone) with an explanation given in Table 4.4. The thin gray lines betoken the elongated cone. The whole cone is covered by the filled upper circle, so no distinction between the upper and lower circle, like with the cylinder, is necessary and thus the silhouette is straightforward:

\[
M \pm \begin{pmatrix} g \\ 0 \\ 0 \end{pmatrix} \pm \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix}, \quad M = \begin{pmatrix} C_x \\ C_y + h \\ C_z \end{pmatrix}
\]
Figure 4.7.: Cone Silhouette (Case 1)

Table 4.4.: Legend Cone Silhouette (Case 1)

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>position of the eye point above the cone</td>
</tr>
<tr>
<td>$E_2$</td>
<td>position of the eye point below the cone</td>
</tr>
<tr>
<td>$C$</td>
<td>origin of the cone</td>
</tr>
<tr>
<td>$g$</td>
<td>radius of the cone</td>
</tr>
<tr>
<td>$h$</td>
<td>height of the cone</td>
</tr>
<tr>
<td>$M$</td>
<td>center of the silhouette circle</td>
</tr>
</tbody>
</table>
b. Case 2 (outside above)

The solution for this case is closely related to the solution for case 1 (Page 40). But taking the upper circle alone is not enough, as can be seen in Figure 4.8

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>position of the eye point</td>
</tr>
<tr>
<td>$E'$</td>
<td>position of the projected eye point</td>
</tr>
<tr>
<td>$C$</td>
<td>origin of the cone</td>
</tr>
<tr>
<td>$g$</td>
<td>radius of the cone</td>
</tr>
<tr>
<td>$h$</td>
<td>height of the cone</td>
</tr>
<tr>
<td>$M$</td>
<td>center of the silhouette quad</td>
</tr>
</tbody>
</table>

**Table 4.5.:** Legend Cone Silhouette (Case 2)

![Figure 4.8.: Cone Silhouette (Case 2)](image)

It has to be ensured that the lower peak of the cone is visible. To accomplish this, the
quad containing the upper circle is rotated in a way, that two corners lie exactly on the straight line $E' + t \cdot \overrightarrow{E'M}$, $t \in \mathbb{R}$, $M = (C_x, C_y + h, C_z)^T$, $E' = (E_x, C_y + h, E_z)^T$, i.e. the quad is directed towards the viewer. The corners of the not rotated quad can be described like in Case 1 with 2 vectors, each of length $g$. If only one vector should be used for one direction, its length has to be $g \cdot \sqrt{2}$ (this comes from Pythagorean theorem). For the two corners along the viewing ray, a vector $\vec{u}$, that in fact is the correctly scaled viewing vector, is used:

$$\vec{u} := g \cdot \sqrt{2} \cdot \frac{\overrightarrow{E'M}}{|\overrightarrow{E'M}|}$$

The other two corners must be on a vector orthogonal to $\vec{u}$ and the up-vector $\vec{y} = (0, 1, 0)^T$, so $\vec{r}$ is introduced:

$$\vec{r} := g \cdot \sqrt{2} \cdot \frac{\vec{u} \times \vec{y}}{|\vec{u} \times \vec{y}|}, \quad \vec{u} \times \vec{y} = \begin{pmatrix} -u_z \\ 0 \\ u_x \end{pmatrix}$$

Now the important change comes: For the frontmost corner, i.e. $M - \vec{u}$, a potential adjustment is necessary. To determine the point $P$, a ray is sent from the eye point to the peak of the cone:

$$P = E + t \cdot \overrightarrow{EC}, \quad t \in \mathbb{R} \quad (4.48)$$

Since $P$ lies in the same plane as $M$ and this plane is parallel to the x-z-plane, $P$ is partly known: $P = (P_x, M_y, P_z)^T$. Now 4.48 can be solved for $t$ when only looking at the y-coordinate:

$$t = \frac{h - E_y}{EC_y}$$

With the point $P$ a new vector for $\vec{u}$, called $\vec{u}'$ is defined: $\overrightarrow{u'} := \overrightarrow{PM}$. But the with this vector constructed quad (that contains the peak) can be smaller than the quad containing the upper circle, which would cause a too small silhouette. This special case can be seen at the topmost cone in the screenshot in Figure 4.9, where the peak is already contained by the quad that was built to contain the upper circle. So the length of the both vectors $\vec{u}$ and $\vec{u}'$ have to be compared and the longer one is redefined as $\vec{u}$ to ensure a consistent naming. Now that all vectors are known, the quad can be built:

$$M \pm \vec{u}, \quad M \pm \vec{r}$$

**c. Case 3 (outside below)**

This case could also be seen of as part of the previous one. The basic principle is the same, only small differences exist because of the fact, that the eye point now is below the cone. The variables $E'$, $M$, $\vec{u}$, $\vec{r}$ and $P$ are defined in the same way. But as can be seen in Figure 4.9, the point $P$ now lies ‘behind’ the cone, so the vector $\vec{u}'$ is defined differently: $\overrightarrow{u'} := \overrightarrow{MP}$ to point in the same direction as $\vec{u}$. It is also not the frontmost corner, that
needs a correction of $\vec{u}$, but the backmost corner, i.e. $M + \vec{u}$. The test (comparison of the vector’s lengths), however, is the same and also the resulting quad is built the same way:

$$M \pm \vec{u}, M \pm \vec{r}$$

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>position of the eye point</td>
</tr>
<tr>
<td>$E'$</td>
<td>position of the projected eye point</td>
</tr>
<tr>
<td>$C$</td>
<td>origin of the cone</td>
</tr>
<tr>
<td>$g$</td>
<td>radius of the cone</td>
</tr>
<tr>
<td>$h$</td>
<td>height of the cone</td>
</tr>
<tr>
<td>$M$</td>
<td>center of the silhouette quad</td>
</tr>
</tbody>
</table>

Table 4.6.: Legend Cone Silhouette (Case 3)
d. Case 4 (outside front, low)

The silhouette is a triangle with the bottom peak as one corner. The other two corners $P_{1,2}$ are on the upper circle and their normals are orthogonal to $\overrightarrow{EP_{1,2}}$ (E is the eye point), which comes from the definition of a silhouette. The schema for this approach can be seen in Figure 4.10. First of all these points are determined, then they are placed in the right order and after that a correction carried out to contain the whole cone.
**Table 4.7.: Legend Cone Silhouette (Case 4)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>position of the eye point</td>
</tr>
<tr>
<td>$C$</td>
<td>origin of the cone</td>
</tr>
<tr>
<td>$r$</td>
<td>radius of the cone</td>
</tr>
<tr>
<td>$h$</td>
<td>height of the cone</td>
</tr>
<tr>
<td>$P_{1,2}$</td>
<td>upper silhouette points</td>
</tr>
<tr>
<td>$T$</td>
<td>front-most top corner of cone</td>
</tr>
<tr>
<td>$V$</td>
<td>plane, in which silhouette lies</td>
</tr>
</tbody>
</table>

**Determination of Points** The points on the outer edge of the upper circle are defined as $P_{1,2} = (x_{1,2}, h, z_{1,2})^T$, where $h$ is the height of the cone. The outcome of this are the
normals (see section b for the derivation of the normal):

\[ \vec{n}_{1,2} = 2 \cdot \left( \begin{array}{c} \frac{x_{1,2}}{r^2} \cdot h \\ \frac{z_{1,2}}{r^2} \end{array} \right) = 2 \cdot \left( \begin{array}{c} \frac{x_{1,2}}{r^2} \\ \frac{z_{1,2}}{r^2} \end{array} \right) \]

The vectors from the eye point to the corners are:

\[ \overrightarrow{EP_{1,2}} = \left( \begin{array}{c} x_{1,2} - E_x \\ h - E_y \\ z_{1,2} - E_z \end{array} \right) \]

Since orthogonality is achieved if the scalar product is 0, \( \overrightarrow{EP} \cdot \vec{n} = 0 \) must be true:

\[ 0 = 2 \cdot x \cdot (x - E_x) - 2 \cdot \frac{r^2}{h} \cdot (h - E_y) + 2 \cdot z \cdot (z - E_z) \]

\[ 0 = x \cdot (x - E_x) + r^2 \cdot \left( \frac{E_y}{h} - 1 \right) + z \cdot (z - E_z) \]  

(4.49)

The points also have to be part of the cone, so putting they must fulfill the implicit representation:

\[ 0 = x^2 - \frac{r^2}{h^2} \cdot h^2 + z^2 = x^2 + z^2 - r^2 \]  

(4.50)

Now there are two equations (4.49 and 4.50) with two unknown variables (\( x \) and \( z \)). To solve this, I first bring them together (there are two terms equal to zero and thus equal to each other) and then transform them slightly (the quadratic summands cancel each other out) to get:

\[ r^2 \cdot \frac{E_y}{h} - x \cdot E_x - z \cdot E_z = 0 \]  

(4.51)

Now a case differentiation is necessary. If the condition \( E_x \neq 0 \) holds, sub-case 1 is the right one, otherwise sub-case 2 has to be chosen.

Sub-Case 1 \( (E_x \neq 0) \)

Equation 4.51 is solved by \( x \) with the newly defined variable \( a \):

\[ a := r^2 \cdot \frac{E_y}{h} \]

\[ x = \frac{a - z \cdot E_z}{E_x} \]  

(4.52)

Inserting \( x \) into 4.50 yields:

\[ 0 = \frac{(a - z \cdot E_z)^2}{E_x^2} - r^2 + z^2 \]

Which can be transformed to:

\[ 0 = z^2 \cdot (E_x^2 + E_z^2) - 2 \cdot a \cdot z \cdot E_z + a^2 - r^2 \cdot E_x^2 \]  

(4.53)

Now again some variables are introduced:

\[ b := E_x^2 + E_z^2 \]
\[ p := \frac{a \cdot E_z}{b} \]
\[ q := \frac{a^2 - E_z^2 \cdot r^2}{b} \]

And equation 4.53 is divided by \( b \), resulting in:

\[ 0 = z^2 - 2 \cdot p \cdot E_z + q \]

With the well-known solutions:

\[ z_{1,2} = p \pm \sqrt{p^2 - q} \]

The solutions \( z_{1,2} \) are inserted into equation 4.52 to gain \( x_{1,2} \).

**Sub-Case 2 \( (E_x = 0) \)**

Because \( E_x = 0 \), the previous sub-case would cause a division by zero, which is not allowed. Thus equation 4.51 is solved by \( z \) instead of \( x \), resulting in:

\[ z = \frac{a - x \cdot E_x}{E_z} \quad (4.54) \]

Where \( a \) is defined the same way as in sub-case 1. The following transformation steps are basically the same, so I get, corresponding to equation 4.53, this:

\[ 0 = x^2 \cdot (E_x^2 + E_z^2) - 2 \cdot a \cdot x \cdot E_x + a^2 - r^2 \cdot E_z^2 \quad (4.55) \]

With the same definition of \( b \), but another ones of \( p \) and \( q \):

\[ p := \frac{a \cdot E_x}{b} \]
\[ q := \frac{a^2 - E_z^2 \cdot r^2}{b} \]

The solutions are the same:

\[ x_{1,2} = p \pm \sqrt{p^2 - q} \]

Again the solutions \( x_{1,2} \) are inserted into 4.54 to get \( z_{1,2} \).

**Order of Vertices** Now the points \( P_{1,2} \) are known, but they have to be in anti-clockwise order, otherwise they would be discarded if backface culling is enabled. The problem is, that nothing is known about the position of the points in relation to the eye point (see Figure 4.11, where both \( A \) and \( B \) can be \( P_1 \) or \( P_2 \)). To get this information, the cross product \( \vec{u} \) of \( (C_x - E_x, 0, C_z - E_z)^T \) and \( \vec{P_1P_2} \) is calculated. Since the cross product obeys the right-hand rule, \( A = P_1, B = P_2 \) if \( u_y > 0 \), otherwise \( A = P_2, B = P_1 \). The vector \( \vec{r} \) then is defined as \( \vec{r} = \frac{\vec{u}}{2} \) and always points in the same direction regarding the cone.
Correction  The last problem is that not the whole cone is covered by the silhouette, because the silhouette actually also includes the front-facing upper circle, although there the normals are not orthogonal to the viewing ray, but this is simply the edge of the corner. To also cover this part, the silhouette is projected into the plane $V$, which is defined by $C$, $T$ and the vector $\vec{r}$. $T$ is the front-most corner of the cone on the upper circle, i.e.

$$
T = \left( \begin{array}{c}
C_x \\
C_y + h \\
C_z 
\end{array} \right) + r \cdot \frac{\vec{CE}'}{|\vec{CE}'|}, \quad E' = \left( \begin{array}{c}
E_x \\
E_y \\
E_z 
\end{array} \right)
$$

This projection is simply achieved by translating $M$ (midpoint of $P_1, 2$) to $T$. This is correct, but delivers a silhouette that is larger than the actual cone, because actually $\vec{r}$ could be shortened to cover the same pixels because the translation is in direction to the eye point which is only a point without size. This correction is done in Case 5, because there the eye point is at least a little bit above the x-z-plane, which is important for the approach. In Figure 4.10 this too big silhouette is shown, in contrast to the precise silhouette in Figure 4.12.

Silhouette  The four corners of the silhouette are:

$$
C, \quad C \, T \pm \vec{r}
$$

e. Case 5 (outside front)

The situation is basically the same as in case 4, but a minimum of 0.1 as height of the eye point is assured. The idea behind the approach is the same as in case 4, but this time with a shortened $\vec{r}$. The approach itself is more complicated. First of all, again the two points $P_{1,2}$ on the silhouette are determined, but this time in the same height as the eye point. Then again this is projected into the plane $V$ (see paragraph d), where the shortening of $\vec{v}$
takes place. After that the silhouette is stretched to reach to the upper circle of the cone. Because the eye point’s height is taken, it should be greater than 0, otherwise $M = C$ and no projection / shortening / stretching is possible. The whole process can be followed in Figure 4.12, where all important points are marked.
### Table 4.8.: Legend Cone Silhouette (Case 5)

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>position of the eye point</td>
</tr>
<tr>
<td>$C$</td>
<td>origin of the cone</td>
</tr>
<tr>
<td>$r$</td>
<td>radius of the cone</td>
</tr>
<tr>
<td>$h$</td>
<td>height of the cone</td>
</tr>
<tr>
<td>$M$</td>
<td>midpoint of silhouette line at eye level</td>
</tr>
<tr>
<td>$M'$</td>
<td>$M$ projected onto $V$</td>
</tr>
<tr>
<td>$T$</td>
<td>front-most top corner of cone</td>
</tr>
<tr>
<td>$V$</td>
<td>plane, in which silhouette lies</td>
</tr>
</tbody>
</table>

**Determination of Points** This is basically the same as in case 4, but this time the points are defined as $P_{1,2} = (x_{1,2}, E_y, z_{1,2})^T$, where $E$ is the eye point. Repeating the same transformation steps leads to:

\[
0 = x \cdot (x - E_x) + z \cdot (z - E_z)
\]

\[
0 = x^2 - \frac{r^2}{h^2} \cdot E_y^2 + z^2
\]

\[
\frac{r^2}{h^2} \cdot E_y^2 - x \cdot E_x - z \cdot E_z = 0
\]

Again there are two sub-cases, depending on the eye point. If the condition $E_x \neq 0$ holds, sub-case 1 is eventuated, otherwise sub-case 2 has to be chosen.

**Sub-Case 1 ($E_x \neq 0$)**

Equation 4.58 is solved by $x$ with the newly defined variable $a$:

\[
a := \frac{r^2}{h^2} \cdot E_y^2
\]

\[
x = \frac{a - z \cdot E_z}{E_x}
\]

The next definitions change a little bit:

\[
b := E_x^2 + E_z^2
\]

\[
p := \frac{a \cdot E_x}{b}
\]

\[
q := \frac{a^2 - a \cdot E_x^2}{b}
\]

But the solutions stay the same:

\[
z_{1,2} = p \pm \sqrt{p^2 - q}
\]

The solutions $z_{1,2}$ are inserted into equation 4.59 to gain $x_{1,2}$.

**Sub-Case 2 ($E_x = 0$)**
Because $E_x = 0$, the previous sub-case would cause a division by zero, which is not allowed. Thus equation 4.58 is solved by $z$ instead of $x$, resulting in:

$$z = \frac{a - x \cdot E_x}{E_z}$$

Again the definitions of $p$ and $q$ change:

$$p := \frac{a \cdot E_x}{b}$$
$$q := \frac{a^2 - a \cdot E_z^2}{b}$$

The solutions are the same:

$$x_{1,2} = p \pm \sqrt{p^2 - q}$$

Again the solutions $x_{1,2}$ are inserted into 4.60 to get $z_{1,2}$.

**Order of Vertices** Thereafter the right order of the vertices is guaranteed by the same principle as in case 4 (see Paragraph d) resulting in the vector $\vec{r}$.

**Projection** The projection is still done at the height $E_y$ because then the intercept theorem can be applied. As can be seen in Figure 4.13, the following equation holds:

$$\frac{m_{Old}}{m_{New}} = \frac{r_{Old}}{r_{New}}$$

and thus $r_{New} = \frac{m_{New} \cdot r_{Old}}{m_{Old}}$. Because $\forall a \in \mathbb{R}, \vec{x} \in \mathbb{R}^3 : |s \cdot \vec{x}| = |s||\vec{x}|$ is true, $\vec{r}$ only needs to be multiplied by $\frac{m_{New} \cdot r_{Old}}{m_{Old}}$.

![Figure 4.13: Cone Projection](image-url)
**Stretching**  Again the intercept theorem is useful here. Figure 4.14 shows the situation when looking towards the cone. The interesting value $r_{\text{New}}$ can be calculated: $r_{\text{New}} = \frac{h_{\text{New}}}{h_{\text{Old}}} r_{\text{Old}}$. Because the silhouette is in plane $V$, the length of the vectors in $V$ are the right ones, i.e. $h_{\text{New}} = |\overrightarrow{CT}| = \sqrt{r^2 + h^2}$ for example.

![Figure 4.14.: Cone Stretching](image)

**Silhouette**  Finally the four vertices can be specified:

$$C, \ C T \pm \vec{r}$$

### 4.3.2. Intersection Shader

No incremental solution (at least not in the sense of the incremental variants before) is possible for the cone intersection shader. So only an explicit solution, that is always applicable, is implemented.

**a. Solution (explicit)**

There is no additional input from the vertex shader needed here, only the usual position of the eye point $E$ and the ray vector $\vec{s}$. Like always, the beginning is made with two formulas for the ray and the cone:

$$E + \lambda \cdot \vec{s}, \lambda \in \mathbb{R} \quad (4.61)$$

$$R_x^2 - \left( \frac{g}{h} \cdot R_y \right)^2 + R_z^2 = 0, \ 0 \leq R_y \leq h \quad (4.62)$$

As already mentioned in the cone silhouette shader, $g$ is the radius and $h$ the height of the cone. Bringing them together results in:

$$(E_x + \lambda s_x)^2 - \left( \frac{g}{h} \right)^2 \cdot (E_z + \lambda s_z)^2 + (E_z + \lambda s_z)^2 = 0 \quad (4.63)$$
Then some intermediate values are defined:

\[
v := \frac{g}{h}
\]

\[
s := s^2_x - v^2 \cdot s^2_y + s^2_z
\]

\[
p := E_x \cdot s_x - v^2 \cdot E_y \cdot s_y + E_z \cdot s_z
\]

\[
q := E_x^2 - v^2 \cdot E_y^2 + E_z^2
\]

The value \( q \) comprises the information, if the camera is inside the elongated cone. To check, if the eye point is inside the cone itself, also \( 0 \leq E_y \leq h \) must be true. The shortened form of 4.63 is:

\[
\lambda^2 \cdot s + 2 \cdot \lambda \cdot p + q = 0
\]

Before starting with the normal cone intersection, one special case is handled. When \( \vec{s} \) is parallel to one side of the cone, the intersection can only be between \( \vec{s} \) and the other side of the cone (see Figure 4.15). This is the case, if the ratios of the width to the height are the same, i.e. \( \frac{g}{h} \). The width and height of \( \vec{s} \) are \( \sqrt{s^2_x + s^2_z} \) and \( s_y \). To avoid computing the square root, the squares of the ratios are compared:

\[
\frac{g^2}{h^2} = \frac{s^2_x + s^2_z}{s^2_y}
\]

If this is true, \( s = \left(s^2_x + s^2_z\right) - v^2 \cdot s^2_y = n \cdot g^2 - \frac{v^2 \cdot n \cdot h^2}{h^2} = n \cdot g^2 - g^2 \cdot n = 0, n \in \mathbb{R} \). So equation 4.64 is only linear:

\[
2 \cdot \lambda \cdot p + q = 0
\]

And has the solution

\[
\lambda = -\frac{q}{2 \cdot p}
\]

The resulting point \( P = E + \lambda \cdot \vec{s} \) is tested agains the additional constraints \( 0 \leq y \leq h \) and discarded, if one constraint is not fulfilled.
Otherwise (if equation 4.65 is false), 4.64 has the obvious solutions:

\[
\lambda_{1,2} = -\frac{p}{s} \mp \sqrt{\frac{p^2}{s^2} - \frac{q}{s}}
\]  
(4.67)

This time two intersection points emerge: \( P_{1,2} = E + \lambda_{1,2} \cdot \vec{s} \). The additional constraints are also checked for \( \lambda_1/\lambda_2 \), but again the manual z-clip (see chapter 3.4.1) is performed for \( \lambda_1 \), which can lead to taking \( \lambda_2 \) or even discarding the fragment. It is possible, that both \( \lambda_1 \) and \( \lambda_2 \) do not fulfill the additional constraints, because the silhouette quad is bigger than the actual silhouette.

Two vectors of dimension three containing \( E \) and \( \vec{s} \) and the constant (for one cone) value \( q \) are needed as input from the vertex shader.
b. Normal Computation

As usual, the normal is the derivation of the implicit surface representation, that is obtained by the help of the ∇-operator:

\[ \vec{n} = \nabla \left( x^2 - v^2 \cdot y^2 + z^2 \right) = (2 \cdot x, -2 \cdot v^2 \cdot y, 2 \cdot z) \begin{pmatrix} P_x \\ -v^2 \cdot P_y \\ P_z \end{pmatrix} \triangleq 2 \begin{pmatrix} P_x \\ -v^2 \cdot P_y \\ P_z \end{pmatrix} \]

Like in the cylinder intersection shader, the pre-computed vectors for replacing the matrix-vector multiplication are modulated:

\[ ep'_{n} = N \cdot \begin{pmatrix} E_x \\ -v^2 \cdot E_y \\ E_z \end{pmatrix} \]
\[ rv'_{n} = N \cdot \begin{pmatrix} s_x \\ -v^2 \cdot s_y \\ s_z \end{pmatrix} \]

The constant factor 2 is, of course, left out:

\[ \vec{n} = N \cdot \begin{pmatrix} P_x \\ -v^2 \cdot P_y \\ P_z \end{pmatrix} = N \cdot \begin{pmatrix} E_x + \lambda \cdot s_x \\ -v^2 \cdot (E_y + \lambda \cdot s_y) \\ E_z + \lambda \cdot s_z \end{pmatrix} = ep'_{n} + \lambda \cdot rv'_{n} \]

N is again the normal matrix. As already stated in 3.4.3, normalisation is not necessary in the intersection shader.
5. Framework

Since the shaders for the different types of geometric primitives should be used inside any OpenGL application easily, a white-box framework was designed to be rather easy to use, but also flexible. A reduced class diagram can be seen in Figure 5.1. The core class is GeometricRenderer, from which an own class must be inherited to specify the positions of the light sources and do some initial code (if desired). This class owns some other classes:

- a Camera instance for specifying the position and viewvector of the viewer
- a ShaderLibrary instance that actually contains the shaders for rendering the primitives and methods to call them
- a DeferredShader instance that is responsible for the deferred shading phase
- some Scene instances that contain code for drawing a scene with the help of the library and normal OpenGL calls
The filenames of the deferred shaders are given as parameters to the \texttt{init} method.
Not in the class diagram is the class \textit{PrimitiveRenderer}, which is an general abstract class for rendering primitives with this technique.

### 5.1. GeometricRenderer

A class with this class as superclass must be defined. Then one instance is created and used for the control of the framework. Two methods have to be overwritten:
• *initGLState*: here some initializations can be done (most likely activating certain light sources and setting material and light source color properties)

• *positionLights*: this method is called every time the camera position or viewvector changes so that the positions of the light sources are fixed in respect to world space, not eye space; i.e. when the camera moves ahead, the light sources ‘move’ backwards, so that they have the same position relative to the objects in the scene.

Three methods correspond to methods that exist in nearly every OpenGL environment: *init* for the initialization, *render* for the drawing of the scene and *changeSize* if the size of the OpenGL widget changes. They have to be called manually from the windowing framework, so it works with different frameworks together (e.g. GLUT, FLTK).

A list of scenes is maintained and can be queried as well as changed. Also a camera is hold internally, to which calls can be forwarded instantly or by setting start of movement together with speed and later than end of movement (e.g. *startFlying(speed)* and *endFlying()*). This kind of movement is then performed every time the scene is drawn, resulting in a frame-rate dependent movement speed.

The renderer can also be set to performing a deferred phong shading on all the objects that are drawn using normal OpenGL calls - *setPhongShading(phongShading)*. Additionally a method *getFPS()* returns an approximation of the current frame per second rate based on the last time needed to render the current scene.

A new renderer for a primitive type can be added by calling *addPrimitiveRenderer* which is delegated to the library.

### 5.2. Camera

The Camera class provides methods internally holds the current position and viewvector. It offers methods to get and set these values. Also some useful helper functions are provided:

• *move*: moves the viewer along the viewvector

• *strafe*: moves the viewer on a vector orthogonal to the viewvector and the up-vector \((0, 1, 0)^T\)

• *fly*: moves the viewer along the up-vector

• *lookHorizontal*: changes the horizontal angle of the viewvector

• *lookVertical*: changes the vertical angle of the viewvector

All these methods are standardized in a way, that a call to one of this functions with a parameter value of 1 moves the viewer a distance of 1 in eye space.

### 5.3. ShaderLibrary

The library is the heart of the framework, since it holds the references to the shaders for the primitives and calls them with the right arguments. For every primitive type 4 methods are given: two to draw the object with the origin at the origin of the eye space (e.g. *drawSphere(radius)*) and two methods to translate the origin to a given point in local space (e.g. *drawSphere(radius, xCenter, yCenter, zCenter)*). There are two variants each because one can be used for a single object (e.g. *drawSingleSphere(radius)*) and one is for
use inside a block of \texttt{beginDrawing(primitive)} and \texttt{endDrawing()}, which can be used for faster rendering of multiple objects.

An instance is constructed with a parameter \texttt{onlyOuterSpheres}, that specifies if the implicit variant of the sphere shaders should be used. If this is the case, the rendering is faster, but the result is unspecified, if the camera is anyhow inside a sphere.

A mapping of primitive’s type’s names to renderes for arbitrary types is also managed by the class. A new renderer can be added by calling \texttt{addPrimitiveRenderer} and retrieve one by calling \texttt{getPrimitiveRenderer}.

\section{DeferredShader}

As already described in section (3.4.3), a deferred shader is used for the rendering process. This is encapsulated into the class \texttt{DeferredShader}, whose method \texttt{beginRendering()} is called before the actual rendering and \texttt{endRendering()} after the rendering. In these methods the switch to the second framebuffer and back takes place, as well as the drawing of a quad covering the whole screen to perform the phong lighting for every pixel on the screen.

\section{Scene}

This is only an interface for a simple scene concept. Every scene has a name and an initialization method, where e.g. display list can be built. The main method is \texttt{draw(library)}, where the actual code for drawing the scene is placed. A scene can consist of objects rendered using the \texttt{ShaderLibrary} and also normal OpenGL objects specified through normal OpenGL calls.
6. Performance Analysis

All measurements were taken on a computer with the following configuration:

- AMD Athlon 3200+ processor
- NVIDIA GeForce 6600 GT graphics card
- Windows XP SP2
- OpenGL 2.1 compatible drivers

6.1. General Durations

By using the timer query extension (GL_EXT_timer_query) which is able to measure the pure time it takes to complete some GL commands I got the following duration:

- generation of the display list which gets rendered to actually draw one primitive: 15.065.089 ns ≈ 15 ms
- multiplying the current matrix with a new matrix (add one transformation): 391 ns

6.2. Silhouette Determination

To determine the time needed to calculate the silhouette of an object is not that easy. The problem is that only the time for passing the vertex shader is needed, but you can not send vertices to the GPU without getting processed by the whole pipeline (vertex shader, clipping / culling, rasterization, fragment shader). So an approximation of the time of the vertex shader can be get by cutting the pipeline in the second step by clipping the object. One simple possibility to reach this is to position the object behind the viewer.

Now the time needed to render one single object is needed. So the activation and deactivation of the shader is not part of the interesting code, also building the display list should be done before. To measure the duration I again used the timer query extension (GL_EXT_timer_query). So the process works as follows: activate the shader, start the query, draw a sphere 10.000 times behind the viewer, end the query. With this technique I got the following results:
Silhouette Determination

![Silhouette Determination Diagram]

**Figure 6.1.** Silhouette Determination

<table>
<thead>
<tr>
<th>Shader</th>
<th>Duration per silhouette determination [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere normal</td>
<td>256.9</td>
</tr>
<tr>
<td>Sphere translated</td>
<td>257.4</td>
</tr>
<tr>
<td>Cylinder normal</td>
<td>323.6</td>
</tr>
<tr>
<td>Cylinder translated</td>
<td>316.2</td>
</tr>
<tr>
<td>Cone normal</td>
<td>186.25</td>
</tr>
<tr>
<td>Cone translate</td>
<td>191.25</td>
</tr>
</tbody>
</table>

**Table 6.1.** Performance Silhouette

### 6.3. Intersection Computation

With the help of the tool NVShaderPerf 1.8 (see NVShaderPerf [http://developer.nvidia.com/object/nvshaderperf_1_8_home.html](http://developer.nvidia.com/object/nvshaderperf_1_8_home.html)) the metrics for the most expensive path for the different shaders were obtained. The duration needed to process one fragment is only a rough approximation since it neglects parallelism, because it is only derived from the pixel throughput, where the number of pixel pipes is taken into account.
6.4. Deferred Shading

Again with the tool NVShaderPerf 1.8 I got:

- $4.21 \cdot 10^6$ fragments / second
- about $237.53$ ns takes it to process one pixel

6.5. Rendering time

To calculate an approximation of the time needed to render $n$ frames, the following formula can be used:

$$t_{\text{render}} = t_{\text{dl}} + n \cdot (p \cdot t_{\text{sil}} + f \cdot (a \cdot t_{\text{inter}} + t_{\text{def}}))$$

, where $t_{\text{dl}}$ is the time needed to build the display list, $p$ is the amount of primitives, $t_{\text{sil}}$ is the time needed to compute the silhouette, $f$ is the number of pixels on the screen, $a$ is the average count of fragments per pixel, that emerge from rasterization , $t_{\text{inter}}$ is the
time needed to perform the intersection calculation and \( t_{\text{def}} \) is the processing time of the deferred shader.

The main contribution to the rendering time depends on the scene and on the viewpoint. If many primitives are being rendered, the factor \( a \) gets bigger and thus the intersection shader is the most critical piece of code. The silhouette determination’s performance is not so important because usually \( f \cdot a > p \), but still should be optimized up to a certain point. If the quantity of primitives \( p \) and also the factor \( a \) are very small, the deferred shader can become the main processing time consumer and thus it would be better to renounce it, which also avoids the problems with the deferred shader (see discussion in 3.4.3).

6.6. Shader-Tessellation-Comparison

With the hybrid variants for sphere and cylinder and the explicit one for the cone some measurements were taken. Therefore 6 test scenes were built, 2 scenes for every primitive: a small and a large one. Every scene consists only of objects that have the same type and the objects themselves are arranged in layers parallel to the x-z-plane. Every layer builds a grid of 50 * 50 objects, where the size radii of the objects decrease with every layer (the lowest layer has the highest radius). Every scene is rendered from 7 view points, 7 times per view point. The time needed for the whole process is taken together with the number of frames drawn. From these values the time needed to render one frame is derived. The tessellation was done in two variants: the simpler one only calls the same display (generated by calling the functions \textit{gluSphere} and \textit{gluCylinder}) list for every object with the same size, while the better one chooses from 4 display lists, with different resolutions, depending on the distance to the viewer. Also both variants for the shaders (with and without passing the center) were distinguished.

\begin{center}
\textbf{Shader-Tessellation-Comparison}
\end{center}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{performance_comparison.png}
\caption{Performance Comparison}
\end{figure}
As can be seen, the shaders are normally faster (other tests during the development have showed that), and even if not, they produce much higher image qualities.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Simple T.</th>
<th>Better T.</th>
<th>Shaders normal</th>
<th>Shaders translated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres Small</td>
<td>980.86 ms</td>
<td>186 ms</td>
<td>76.2 ms</td>
<td>59.3 ms</td>
</tr>
<tr>
<td>Spheres Large</td>
<td>1231 ms</td>
<td>472 ms</td>
<td>159.8 ms</td>
<td>67.9 ms</td>
</tr>
<tr>
<td>Cylinders Small</td>
<td>239.8 ms</td>
<td>32.9 ms</td>
<td>56.1 ms</td>
<td>54.5 ms</td>
</tr>
<tr>
<td>Cylinders Large</td>
<td>142.5 ms</td>
<td>80.37 ms</td>
<td>142.6 ms</td>
<td>73.65 ms</td>
</tr>
<tr>
<td>Cones Small</td>
<td>237 ms</td>
<td>33.2 ms</td>
<td>56.1 ms</td>
<td>54.8 ms</td>
</tr>
<tr>
<td>Cones Large</td>
<td>139.7 ms</td>
<td>122.1 ms</td>
<td>143.2 ms</td>
<td>66.65 ms</td>
</tr>
</tbody>
</table>

Table 6.3.: Performance Comparison
7. Limitations and Future Prospects

7.1. Deferred Shading

The second rendering phase could be improved to realize fog, shadows and other sophisticated things. It would also be possible to use more than two textures for storing the properties of the fragments, in order to increase variability, e.g. by also storing the shininess or ambient color of the current OpenGL state. It has to be researched, if a full integration into OpenGL, i.e. mimicking the whole fixed functionality pipeline with all its features, makes sense, i.e. is faster than using the fixed pipeline directly. If this is the case, the deferred shading could be applied to almost any OpenGL application to increase performance.

7.2. Backface Culling

The problem with backface culling is the same as with user clipping (see 3.7): if it is done in the second phase, discarded pixels produce holes, that could be filled if done directly in the intersection shaders. But if it is done in the intersection shaders themselves, an additional parameter holding the information, if backface culling is enabled, has to be transferred and an additional conditional is necessary to decide, if culling is done. A solution to this problem is to write two variants of the shaders: one with backface culling, one without, but both without the additional parameter and conditional. If a primitive should be drawn, the framework decides, which variant is the appropriate one.

This solutions could also be useful to support full user clipping, but then many variants must be written \((2^n, \text{where } n \text{ is the quantity of possible clipping planes})\).

7.3. Texturing

It is not that complicated to implement support of texturing (at least for this 3 types of primitives). Only the texture coordinates must be determined and than stored in an additional texture (as buffer for the second rendering phase). The problems are that the calculation of the coordinates is time-consuming (needs sine and cosine for the primitives covered) and it is very complicated, if not impossible, to track the change of the active textures. A tradeoff is to use the texture that is active when the deferred shader runs for every pixel on the screen.

Further research is necessary to check if and how texture filtering can be combined with this approach.

7.4. More Primitives

Of course also additional primitives can be supported. Only a silhouette and an intersection shader are needed. In most cases the silhouette shader will be more complicated than the intersection shader, at least, if an explicit solution is necessary. But sometimes also this
can become complicated, e.g. if the root finding can not be done analytically, but must be solved by numerical methods.

7.5. Conclusion

The idea of ray-tracing geometric objects on the GPU via shaders has some potential since new primitives can be added and the speed and quality reached is very high. For specialised applications like molecule viewers where other types of objects and also a full support of OpenGL features are not needed it overtops tesselation approaches. But I do not think it has good prospects in other types of applications like games. The biggest problems are those, that arise from the deferred shading, where it is complicated to reach the full flexibility that comes along with direct shading.

Maybe if direct shading becomes faster because of highly optimized GPU commands that are necessary for shading (e.g. dot product) or by adding mechanisms to manipulate OpenGL internals (e.g. normalization of the stored light source vectors, so not every fragment processing unit has to normalize them each time it does the lighting calculation for a fragment).

On the other hand, tesselation can be improved as well. For example with a level of detail the rendering time can be reduced significantly, although still retaining a good image quality.

But the performance of this approach can still be enhanced. If the shaders get all information that are necessary to render a object, i.e. not only the translation, but also the direction vector in case of the cylinder or other such values, the calls to manipulate the current model-view matrix could be decreased to a very small number which gives a big boost to the rendering. The disadvantage of this change is a more complicated usage since the correct vectors have to be computed in some way on the CPU where it is important to check that it does not slow down the process.

Another great benefit could be achieved by improving the silhouette determination even more. For example, the cylinder shader in the case, where the viewer is above or below the cylinder, produces a silhouette quad that is bigger than the actual cylinder because of the projective deformation. Maybe in some cases the silhouette could be approximated by other polygons than quads.

To achieve a better support of the OpenGL functionality, a proposals was made (see section 7.2).
## List of Figures

2.1. Screenshot from [3] (tensor field visualized by large number of ellipsoids) . . . . 7
2.2. Screenshot from [11] (dragon SLIM model with different shaders) ............. 8
2.3. Screenshot from [6] (torus clipped by bounding tetrahedron) .................. 8
2.4. Screenshot from [5] (bar magnet illustrated by 6,000 particles) ............... 9
2.5. Screenshot from [10] (200,000 dipoles) ........................................ 10
2.7. Screenshot from [4] (left: NURBS scene, right: together with 9 NURBS trimming curves) ................................................................. 11
2.8. Screenshot from [7] (ribbon visualization style) .................................. 12
3.1. OpenGL pipeline ................................................................. 14
3.2. Advanced OpenGL pipeline .................................................. 15
3.3. View from inside the cylinder upwards ........................................ 17
3.4. Rays through silhouette-pixels ............................................ 19
3.5. Manual z-Clip ................................................................. 21
3.6. User Clipping Problem ....................................................... 24
3.7. User Clipping Screenshots (left: full support, right: partly support) ........ 24
4.1. Sphere Silhouette ............................................................... 26
4.2. Sphere Intersection ............................................................ 28
4.3. Cylinder Silhouette (inside) ................................................ 33
4.4. Cylinder Silhouette (outside) ............................................. 34
4.5. Cylinder Silhouette Height Problem ....................................... 35
4.6. Look through Cylinder ....................................................... 38
4.7. Cone Silhouette (Case 1) ..................................................... 41
4.8. Cone Silhouette (Case 2) ..................................................... 42
4.9. Cone Silhouette (Case 3) ..................................................... 45
4.10. Cone Silhouette (Case 4) ................................................... 46
4.11. Cone Vertices Order ........................................................ 49
4.12. Cone Silhouette (Case 5) ................................................... 50
4.13. Cone Projection ............................................................... 52
4.14. Cone Stretching ............................................................... 53
4.15. Cone Parallel Ray Vector ................................................... 55
5.1. Reduced Class Diagram Framework ......................................... 58
6.1. Silhouette Determination ..................................................... 62
6.2. Performance Intersection ................................................... 63
6.3. Performance Comparison .................................................... 64
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Legend Sphere Silhouette</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Legend Cylinder Silhouette (inside)</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Legend Cylinder Silhouette (outside)</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Legend Cone Silhouette (Case 1)</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>Legend Cone Silhouette (Case 2)</td>
<td>42</td>
</tr>
<tr>
<td>4.6</td>
<td>Legend Cone Silhouette (Case 3)</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Legend Cone Silhouette (Case 4)</td>
<td>46</td>
</tr>
<tr>
<td>4.8</td>
<td>Legend Cone Silhouette (Case 5)</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Performance Silhouette</td>
<td>62</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance Intersection</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance Comparison</td>
<td>65</td>
</tr>
</tbody>
</table>
Bibliography


