Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Dahlem Center for Machine Learning and Robotics

Simulating Bee Vision:

Conceptualization, Implementation, Evaluation and Application of a Raycasting Rendering Engine for Generating Bee Views

Johannes Polster

Matrikelnummer: 4663344

johannes.polster@fu-berlin.de

Betreuer & 1. Gutachter: Prof. Dr. Tim Landgraf

2. Gutachter: Prof. Dr. Raúl Rojas

Berlin, 21. August 2017

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 21.08.2017

____

Johannes Polster

Abstract

Vision is the basis for complex behaviour of bees such as navigation and foraging. Simulating bee vision can be helpful in exploring this behaviour. In this thesis a bee vision simulation is developed, based on a detailed scientific model of the spatial resolution and field of view of a bee's eye.

Four different aspects concerning bee vision simulation are covered in this thesis. Namely an existing 3D environment is enhanced by extending it with a skydome. The afore mentioned model is implemented. A rendering engine based on the bee's eye model is built that uses raycasting for generating bee views from the 3D environment. Lastly, an existing tool for analysing flight paths tracked by radar is extended, so that it can be used to render movies from the bee's point of view.

Table of Contents

1 Introduction

It is fascinating how animals perceive the world, since perceptions are subjective gateways to reality specific for every living species. Indeed, given the widely differing perception devices across the animal kingdom it has to be assumed that every animal species lives in a completely different “reality”[1]. Simulations are a good method for studying how animals perceive the world. For studying vision, the physiological properties of an optic system can be examined, however the simulation thereof gives a better understanding. For this reason animal vision simulations are often used in an educational context [1]–[3].

Bee vision, relying on a completely different optical system than human vision, is especially interesting. The simple vision system of the bees is the basis for complex behaviour like navigation and foraging.

Since vision is the main sensory input a bee receives, it is important to have an accurate model of bee vision as the input for neural models. [4], [5]

In this thesis a bee vision rendering engine is presented that can simulate bee vision in a virtual 3D environment. The engine can be used as input for an artificial neural agent, or for further studying the navigational behaviour of bees. This thesis is split into 4 major parts. Each part contains an implementation section that describes the implementation details and an evaluation section in which the results are evaluated.

Virtual environment

In this work, an existing 3D model of a test site in Hesse is extended to include everything one can see form a given point of view. For this a skydome with a 360° panorama as texture is developed and added to the current model.

Bee’s eye model

A model of the spatial resolution and field of view is implemented based on the physiological properties of a bee’s eye.

Renderer

The main task of this thesis is to create an application that is capable of rendering bee views in the virtual environment based on the bee’s eye model.

1 Some philosophers even define reality by what one can perceive: “perception is reality”.

Radartrack movie maker

In order to visualize bee perception for human eyes a tool is built for creating bee-vision-movies from flight paths of bees that were tracked with radar. For this the existing radartrack python package of the *FU Biorobotics Lab* was extended.

1.1 Bee's Eye Basics

In the following chapter I will present the biological background for the bee vision renderer.

Insect vision has fascinated scientists for at least 300 years, because it is so fundamentally different from human vision. The compound eyes of insects are made up of thousands (in the case of the honeybee about 5500) of hexagonally shaped ommatidia facing in different directions. [6]

These ommatidia each act like single eyes with their own corneal lens and (in reference to the apposition eye) photoreceptors. But unlike human eyes each ommatidium receives light from a very limited portion of the environment which results in one picture element (pixel). These single pixels are then stitched together by the brain, a process in which the high temporal resolution of insect eyes (100 frames per second [fps] in case of the honeybee

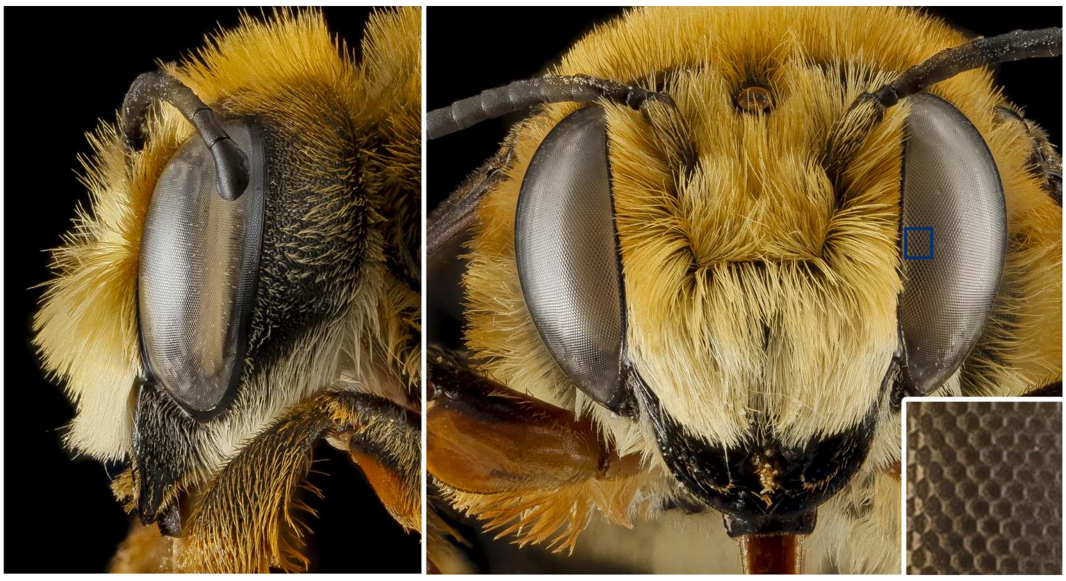


Illustration 1.1: Close up photographs of a Megachile Fortis bee's head from the side (left) and from the front (right). The individual hexagonally shaped facets (the box on the right shows the magnified area of the blue rectangle) and the central ocellus on top of the head can be distinguished. The compound eyes have an ellipsoid shape and are roughly parallel to each other. The strong curvature of the eye makes for a large field of view (FOV): the honeybee has close to a 360° FOV, only limited by the region blocked by the thorax of the bee. Photographs courtesy of the U.S. Geological Survey.

[7]) probably plays a crucial role (after all, each image has to be analysed in the context of the previous images). [8]

Honeybees also perceive a different spectrum of light. While human spectral sensitivity peaks at wavelengths for red, blue and green light, honey bees have sensitivity peaks at ultraviolet, blue and green wavelengths. The ommatidia in the dorsal rim area are also sensitive to the polarization of light, which is useful for navigation. [9]

The bee also has 3 other eyes located on the top of their head: these so called ocelli are small single lens eyes with which the bee can measure smallest changes in light intensity; this is useful to determine the sun's location.

The most relevant physiological parameters for this thesis are the spatial resolution and the field of view of the bee's eyes. These are governed by the following parameters:

Interommatidial angles

The interommatidial angle ($\Delta\varphi$) is the angle of neighbouring ommatidia. One can differentiate horizontal (elevation, $\Delta\varphi_h$) and vertical (azimuth, $\Delta\varphi_v$) angles.

The angle varies across the bee's compound eye, with a minimum at the equator and gradual increments towards the borders of the eye. This makes for smaller spacing of the viewing directions and thus a higher resolution at the equator.

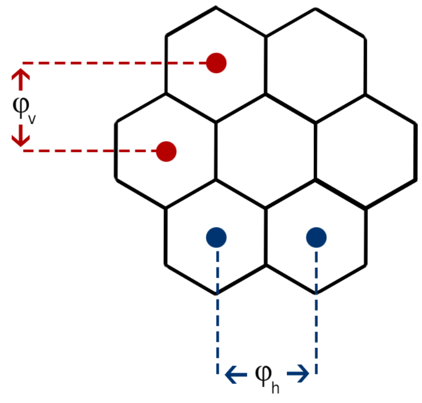


Illustration 1.2: The definition of Δφh and Δφv .

Acceptance angle

The acceptance angle ($\Delta\rho$) defines the visual field of the individual ommatidia.

Acceptance function

The acceptance function describes the sensitivity of the ommatidium, in relation to the angular distance from the optical axis. The angular sensitivity function can be approximated by a two dimension circular Gaussian function. The full width at half maximum (FWHM) of this function is the acceptance angle of the ommatidium. [10]

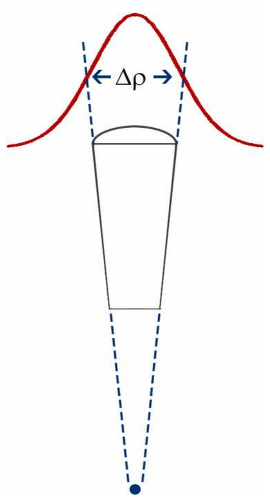


Illustration 1.3: An ommatidium, its acceptance angle (Δρ) and the Gaussian shaped acceptance function (red curve). The blue dot is the centre of the eye.

It is worth noting that even though human vision also relies on the discrete input we do not see the world pixelated. Similarly, one can assume that bees don't see the world in pixels but rather rely on brain processes that smooth the visual input into a continuous image.

Nevertheless, the spatial resolution and field of view put a limit on what the bee can possibly see and are important determinants of the final perception as the sensory input is the first step of visual processing.

In the following chapters *bee view* will refer to the single image a bee sees with its bilateral compound eyes when looking from a certain position in a certain direction (or, better speaking, a model thereof, after all we are not able to fully describe the bee's visual impressions).

1.2 Computer Graphics Basics

In this chapter I will briefly cover some basics and terminology of computer graphics that are used in this thesis.

Rendering. In the context of computer graphics rendering refers to the process of converting 3D models into 2D images (for example rendering is needed to display a 3D object on a computer screen).

Scene. In the context of this thesis, scene refers to the 3D model to be rendered. The scene is made of multiple objects, such as spheres, planes, and meshes. Meshes consist of polygons (usually triangles or quads) called **faces.** The direction of the faces is given by their **normal** vector. The normal vectors are important for shading the objects. In case of triangle meshes the faces are defined by 3 **vertices,** which are points in 3d space. The position of the vertices is given by 3D coordinates.

Objects can have textures. Textures are images that are wrapped around the object via texture mapping. Texture mapping is the process of mapping the pixels of a 2D image onto a 3D model. This is done by assigning texture coordinates to the vertices of the object. The texture coordinates map the 3D vertex coordinates to the 2D coordinates of the texture image. In this context barycentric coordinates play an important role. Barycentric coordinates describe the position of a point in relation to a simplex such as a triangle.

Rays are used to simulate how light travels in a scene. A ray is defined by a point in space and a direction. The ray is normalized to a length of 1. In the context of raytracing primary rays are the first rays cast from the camera into the scene. Whereas secondary rays are cast from the intersection point of the primary rays, e.g. to calculate shadows or reflections. This is also the main difference of **raytracing** and **raycasting**. Raycasting only takes into account the primary rays, whereas raytracing “traces” the primary rays recursively and takes into account multiple bounces of the ray.

Instead of simulating the light omitted from the objects in the scene, only the light that reaches the camera is simulated. To achieve this a backwards approach is followed: rays are generated from the camera and not from the object. For each pixel of the image to be rendered, ray directions are calculated. The rays are “shot” in the calculated directions, by testing if any objects of the scene are in the path of the ray. For this every object in the scene is tested against its intersection with the ray. This is computationally expensive since there can be millions of objects in a scene. To speed this up, the scene can be stored in acceleration data structures. A common data structure used is a bounding volume hierarchy (**BVH**). Objects are wrapped in bounding volumes, bounding volumes close to each other are wrapped again in a bigger bounding volume and so on. Through this wrapping process, a hierarchy of bounding volumes is created. The resulting hierarchy is usually stored in an oct-tree datastructure.

After an intersection is found the colour for the pixel is calculated, depending on the properties of the object (e.g. the texture) and the hitpoint position.

Pitfalls when designing a 3D Graphics Application

There are different ways of defining the direction of the axis of a 3D coordinate system. In this thesis the “right handed” convention is followed, with y defining the up direction. The handedness also has an effect on the cross product. Also, when loading an object that was created with an application using a different convention, its coordinates have to be transformed, or else the object is flipped and the texture is not applied correctly.

In 3D space, objects can be transformed by applying a transformation matrix to the object’s 3D coordinates. Depending on whether matrices are defined in row major or column major order, the transformation matrix has to be applied in a different order. In this thesis the matrices are stored in column major order, so for applying two transformations T1 and T2 to a point P, one needs to use the order: T2 * T1 * P

When designing a 3D application it is important to avoid a problem called gimbal lock. Gimbal lock can occur when using Euler angles for rotating an object. For example, it can occur when rotating a direction vector around an axis of the coordinate system if this vector is aligned with the axis. Solutions to this problem are to change the order of Euler rotations when gimbal lock is detected, or to store the rotation parameters as a single rotation matrix or as quaternions.

2 Related Work

The visual field and spatial resolution of bees, which form the basis of the model eye in this thesis, has been mapped out by Seidl et al [11].

The model for generating the interommatidial angles used in this thesis has been developed by Stürzl et al [12]. It is an extension of an earlier model by Giger [13]which only covers the frontal hemisphere of the eye (180° horizontal field of view). Stürzl's model covers the full FOV of the compound eye, taking into account the boundaries of the eye's visual field. A detailed description is provided in the chapter "Bee's Eye Model".

Based on his model Giger developed an application called the Bee Eye Optics Simulation (BEOS). It simulates how bees perceive 2D images placed at a specific distance from the bee. [14]

Stürzl also developed an application [15]for remapping images to bee-views,. However, the source code is not open source, and the online service provided is limited to the two virtual environments recorded in Australia. These virtual environments also lack a horizon panorama which probably plays a substantial role in honeybee navigation [16]. Other disadvantages include an inconvenient online service which returns resulting images via e-mail. Additionally the input over files makes for an inflexible use of the service. For example, it is difficult to place the bee at a specific height, because the ground height is not known (z ranges from -105 to 500). As of the time of writing this thesis the service for mapping panoramas to bee views does not work.

In Stürzl et al [12] the method for constructing the bee views is described as a remapping of 280° images. A lot of interpolation needs to be used for the remapping, if the resolution of the panoramic images is not very high. The raytracing technique used by the renderer in the present thesis provides a more accurate simulation as it directly simulates the light rays without the need of remapping.

There are several other papers with different attempts to simulate bee vision. Collins [17] developed a Monte Carlo raytracing simulation for honeybee vision. He takes the geometry of the eye as a basis for calculating viewing directions, and also simulates the spectral sensitivity of the eye.

Another bee's eye simulation based on the work of Stürzl et al is a virtual reality game programmed with the Unity game engine. This game is intended for educational purposes, and covers only a small part of the FOV of bees. [3] Another educational game for simulating bee vision is described in [2].

There are also hardware simulations of how a bee perceives the world. In Vorobyev et al [18] the spectral sensitivity of the receptors of an eye is simulated by recording images with a UV-sensitive camera. The spatial resolution is simulated by reconstructing the bee views depending on the interommatidial angles and acceptance angle of the ommatidia.

In [19] Neumann describes a method for mapping a cubic environment map to spherical images with the resolution of insect vision. Written in 2002 he states: “ray tracing is not supported by current computer graphics hardware and is therefore inefficient and slow for complex scenes and large numbers of samples”. Now, 15 years later, with the advancement in hardware and efficient ray tracing algorithms, this claim does not stand up any more, as shown in this thesis.

3 3D Environment

The 3D model was created in June 2016 by the *FU Biorobotics Lab* from aerial images taken by a drone, using stereophotogrammetry. Photogrammetry can be used to reconstruct 3D data from 2D images by comparing images of the same object from different viewpoints. The resulting model has a vertical accuracy of 30 cm and a horizontal accuracy of 5-10 cm. Therefore small bushes and trees are distinguishable but smaller objects such as fences and small plants are only visible in the texture of the model.

The Model was created from an area in Hesse, central Germany, near the town of Marburg. The 3D model covers about 2 square kilometres. The area is a testing site that is used for a multitude of experiments with insects, since there is a harmonic radar instalment similar to Riley [20] that can track bees up to a range of 900m.

Since the model only has an extent of 2 square kilometres, it doesn't cover the surrounding hills and other landmarks. Yet, for some navigational experiments, the current horizon panorama plays an important role, thus the model needed to be expanded as part of this work to cover everything one can see from the ground.

Nine high resolution 360° panoramas taken from different positions in the model (at the end of August 2016) were provided by the Biorobotics Lab. Unfortunately the positional data (the locations from where the panoramas were taken) was lost and had to be reconstructed.

Several possibilities for extending the model were considered:

- Using additional areal drone recordings to calculate a model that covers the surrounding area. The problem of this approach is that creating the model is computationally expensive (the existing 2 square kilometres took 45 days to calculate), of course this puts limits on how large the model can be. Additionally the performance of raytracers depends on the number of faces in the scene, so it is desirable to keep the number of faces at a reasonable level. Another problem is the German law: Firstly the drone always has to be in sight and you are not allowed to fly over the nearby federal highway (Bundesstraße). Secondly it is forbidden to use drones near airports, yet, unfortunately there is one close to the nearby Amöneburg.

- Using the existing panoramas to construct a 3D model with data from spherical photogrammetry that covers the surrounding area. The problem of this approach is that the angle of the rays that meet at the object to be reconstructed is not large enough for far away objects. [21]

- Using elevation data generated from NASA's Shuttle Radar Topography Mission (SRTM) [22]. The problem of this approach is that the data was sampled with an accuracy of 30m and only describes the elevation of earth's surface, the resulting digital elevation model (DEM) therefore cannot be very accurate.
- Projecting one of the panoramas onto a cylinder or sphere and use this as a skydome for the model. The problem of this approach is that the horizon panorama changes depending on the view point of the observer.

I chose the skydome approach because it is the most straightforward. However, I had to tackle the problem of changing horizon depending on the view point of the observer (for details see the evaluation part of this chapter).

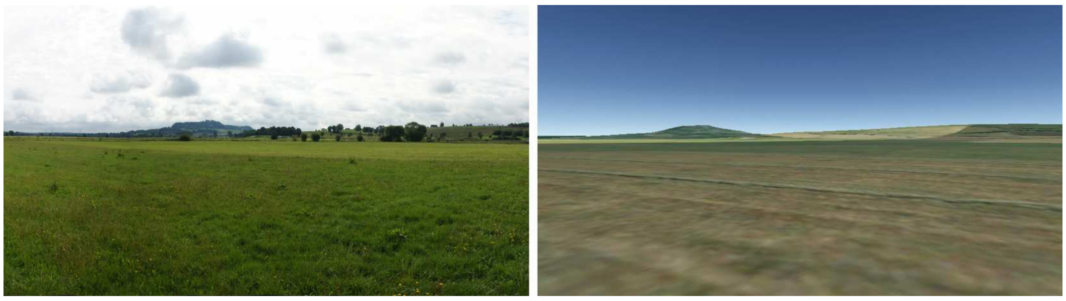


Illustration 3.1: This is how the DEM approach might look, when comparing Google Earth data with the panorama. The DEM results in "flat" vegetation and houses since it only measures the elevation of the surface of the earth. So while this method could be used for the horizon panorama it is not suited for closer objects.

Google Earth image copywrite: 2017 AeroWest, Landsat/Copernicus, 2017 DigitalGlobe, 2017 GeoBasis-DE/BKG.

3.1 Implementation

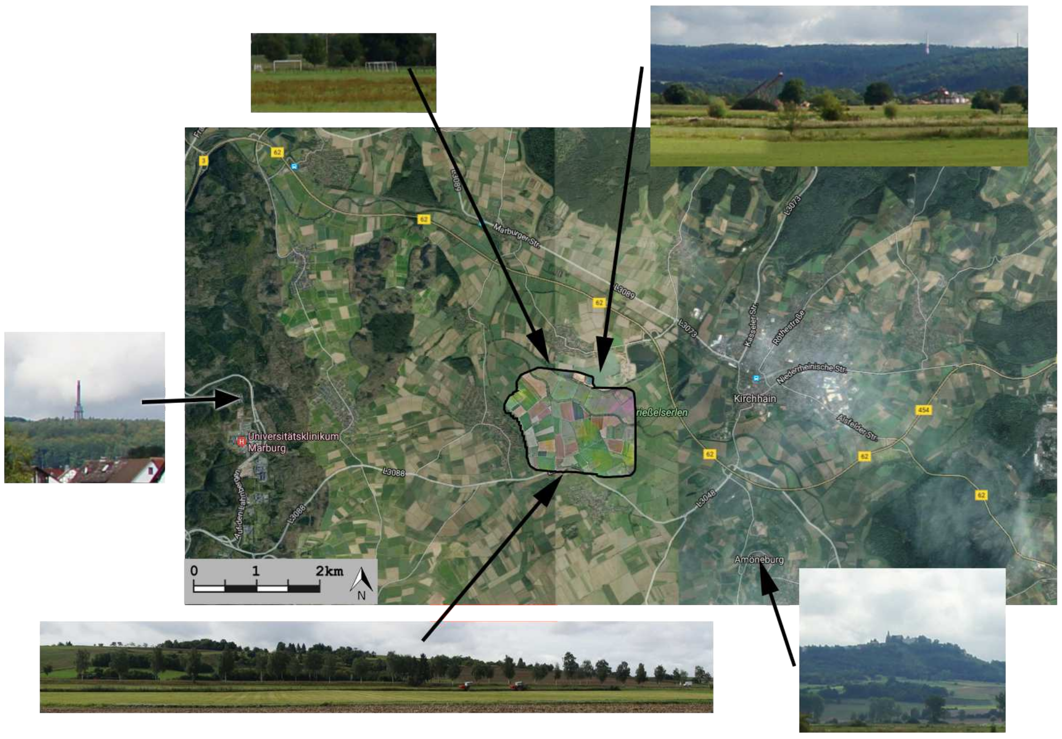


Illustration 3.2: The georeferenced model and its surroundings. This image shows the extent of the model and important landmarks in the panorama. The landmarks were used to accurately align the skydome and to reconstruct the position from where the panorama had been taken. The original capture position is important to determine which panorama is the most central, so this panorama can be used as texture for the skydome.

Google Maps copywrite: Imagery 2017 AeroWest, Landsat/Copernicus, 2017 DigitalGlobe, 2017 GeoBasis-DE/BKG, Maps data: GeoBasis-DE/BKG, Google.

For creating the skydome Blender [23], an open source 3D modelling and rendering package was used. A cylinder surrounding the 3D model was created and the most central panorama was used as texture. Choosing the central panorama for the skydome has the effect that the deviations depending on view point are minimized over the whole scene.

A cylinder was chosen over a sphere since the EXIF data of the panorama indicates that it had been stitched together as a cylindrical projection.

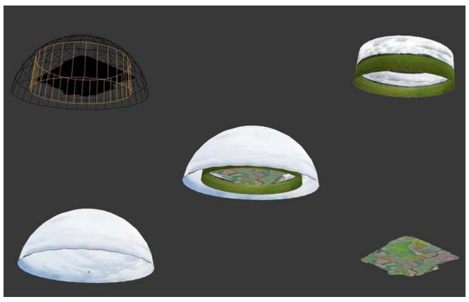


Illustration 3.3: Illustration of how the 3D model is extended. The cylinder with the panorama projected onto it, plus, the hemisphere with the sky texture. Since the sky is not that important and may distract from the panorama I also created a 3D model with solid white sky.

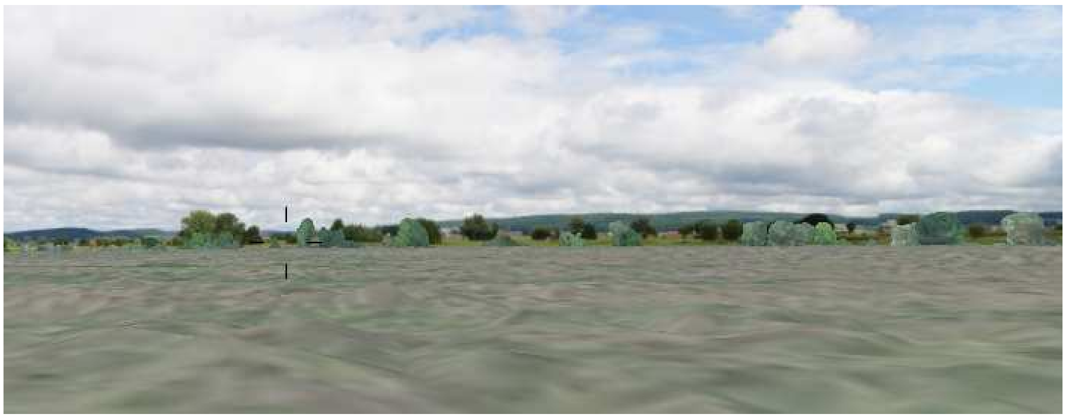


Illustration 3.4: Here the problem of duplicated objects is illustrated. Objects in the 3D model also appear on the panorama. The bushes in the background are duplicated.

Changing the projection to project the panorama on a sphere would have been possible but would have resulted in a loss of resolution.

To correctly align the panorama the cylinder was rotated until objects that are both in the panorama and the 3D model – such as bushes and trees – overlap (compared from the view point of where the panorama was taken, this is also the centre of the cylinder). The cylinder was then scaled until these objects were the same size. For example images see the evaluation section of this chapter.

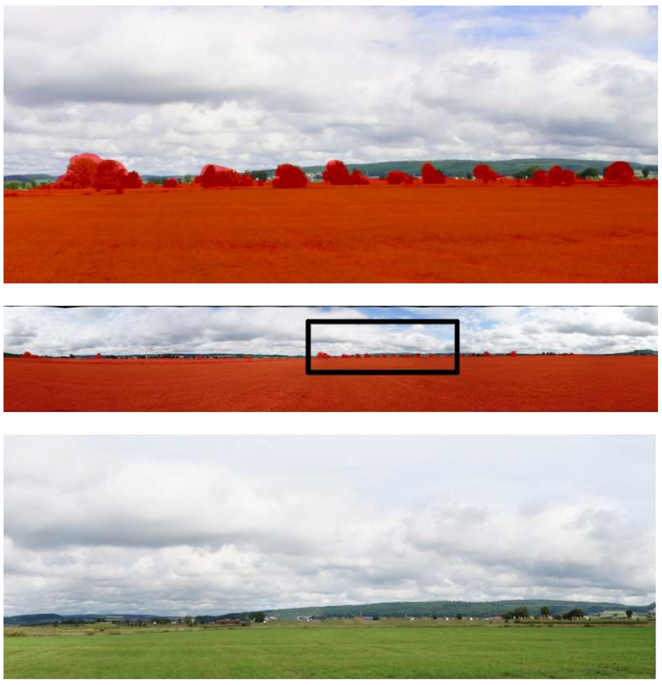


Illustration 3.5: The second image shows the whole panorama. The first and the last image show a zoomed portion of the panorama (from the black box of the second image).

To solve the problem of duplicated objects in the 3D model and the panorama the duplicated objects were identified (upper image, red objects) and removed from the panorama with Adobe Photoshop. Larger objects were replaced with parts of other panoramas, since one can not see what is behind these objects. The lower image shows a portion of the resulting panorama with objects removed.

3.2 Evaluation

The resulting model has 1 000 294 faces and 499 116 vertices. The resolution of the texture of the 3D terrain is 8000×8000px, the resolution of the texture of the cylinder is 24000×3000px. The model needs 472 MB of disk space.

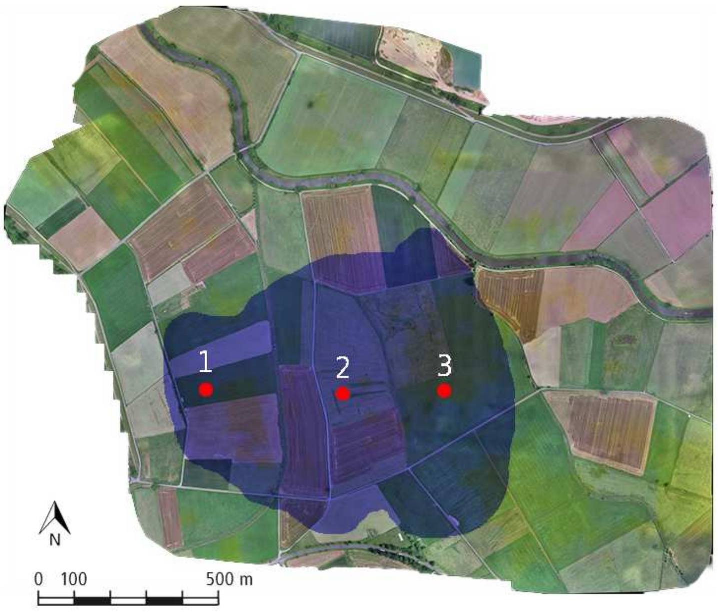


Illustration 3.6: Reconstruction of the original recording positions. The central panorama from viewpoint 2 is used for the skydome. The other two panoramas are used to compare the deviations that arise when looking at the panorama from a different view point (compare figure 3.8). The blue area is the approximate area that the bees covered in the catch-and-release experiments conducted in 2013.

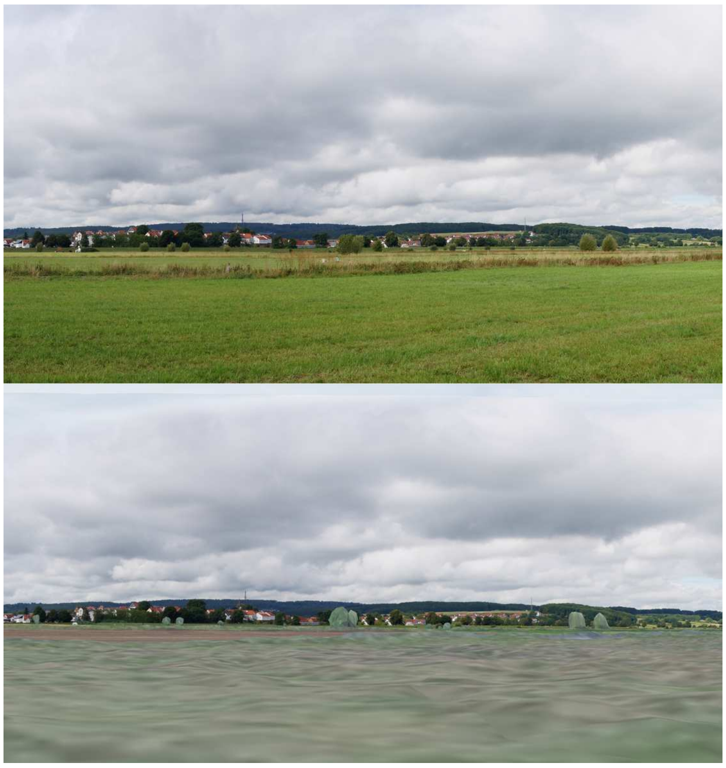


Illustration 3.7: To validate the alignment of the skydome, the sections of the 2D panorama (upper images) are compared with renderings of the 3D model (lower images) from the same viewpoint from which the panorama had been taken (viewpoint 1). The images show that the objects (such as bushes) are approximately at the same position and similar in size.

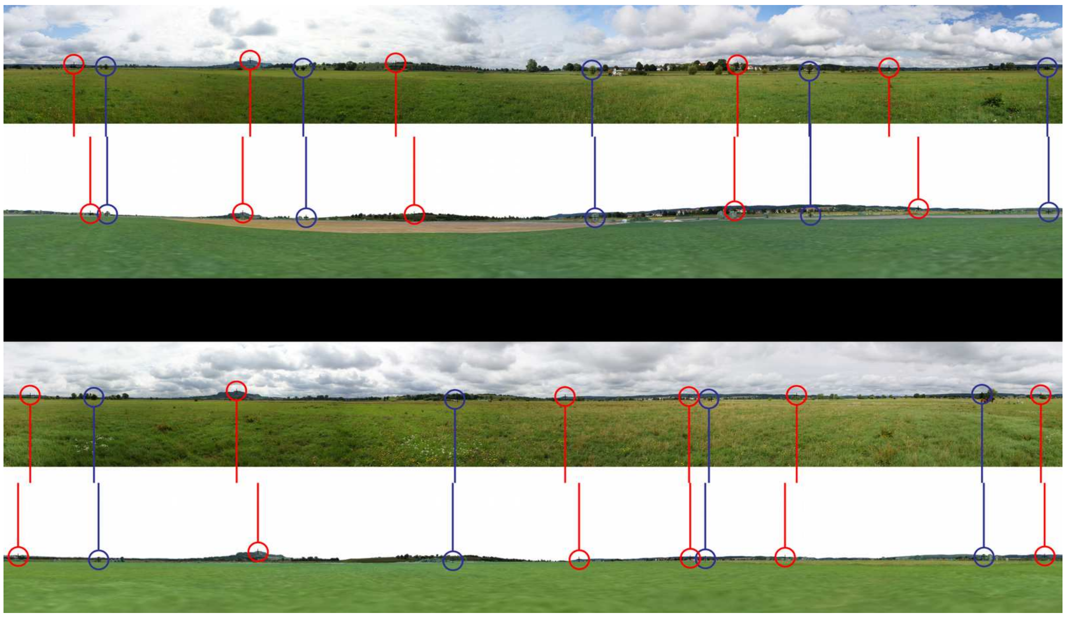


Illustration 3.8: For physical reasons the horizon panorama changes, depending on the position of the viewer. Since a fixed panorama was used that does not change depending on viewpoint the horizon will not show the correct position and size of scene objects. How large the differences are depends on the distance to the objects in the panorama: closer objects will display a larger deviation than far away objects. When moving closer to objects, they become larger, when moving away they become smaller, when moving parallel to the objects they shift. In a 360° Panorama all of these effects can be observed.

The upper figure shows the deviations from viewpoint 1 (upper half is the panorama taken from viewpoint 1, lower half is a rendering of the 3D model from the same position). The lower figure shows the deviations from viewpoint 3.

The blue markers are control markers to show that the rendering is from the same position: If so, objects in the scene have to align with objects in the panorama. The red markers are placed on the same features in the panorama and the texture-panorama. As expected, there are noticeable shifts between the objects depending on the viewpoint.

The sizes of objects change depending on viewpoint. This effect is preserved, since the objects appear bigger with closer distance to the skydome. For example, the village is closer from viewpoint 1 than from viewpoint 3. Both in the panoramas and texture-panoramas the village appears larger from viewpoint 1 than from viewpoint 2.

In summary, it can be stated that while there are deviations from what one would actually see, the deviations observed are not too large.

4 Bee’s Eye Model

In this chapter a model bee’s eye is presented that has been constructed based on the biological background from chapter “Bee’s Eye Basics”. The model describes the spatial resolution of the honeybee’s eye.

4.1 Implementation

As mentioned earlier the spatial resolution is defined by three parameters: the interommatidial angles, the acceptance angle and acceptance function. Here I describe how the parameters are modelled.

4.1.1 Interommatidial Angle

The interommatidial angles vary across the extent of the eye. Vertically they range from 1.5° to 4.5° and horizontally they range from 2.4° to 4.6°.

For calculating the interommatidial angles a routine described by Stürzl et al [12] was implemented. The routine is based on a formula from Giger [13]. Giger approximates the measurements of interommatidial angles gathered by Seidl [11] for all ommatidia in the frontal hemisphere with a numeric method that produces angles in accordance to the measurements. Stürzl et al extend this model to cover the full bee’s eye FOV. They also take into account the border of the visual field.

The following pseudo code gives a brief description of the routine for calculating the interommatidial angles. The exact rules can be found in the appendix of [12]. The routine produces elevation and azimuth angles that describe the angular distance from the bee’s viewing direction (0° elevation and 0° azimuth). Elevation angles are in a range of -90° to +90°, azimuth angles are in a range of -90° to +270°.

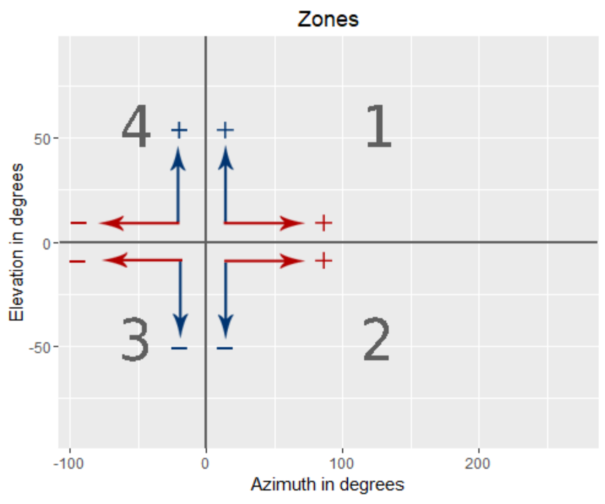


Illustration 4.1: There are four zones. Depending on the zone the angles are either added or subtracted to elevation (ε) and azimuth (α). For each zone different rules for calculating the interommatidial angles apply. The blue arrows illustrate the creation of elevation angles. The red arrows show the row-wise creation of the azimuth angles. The sign at the end of the arrows indicate if the angles are subtracted or added in the routine.

```
for_each zone:
    α = 0, ε = 0, counter = 0
    while |ε| <= 90:
        while |α| <= max_α(ε, zone):
            Δφₕ = calc_Δφ_h(ε, α)
            d = calc_offset(Δφₕ, ε)
            α = α + d
            storage[counter] = (α, ε)
            counter = counter + 1
        end_while
        ε = ε + calc_Δφ_v(ε)
    end_while
end_for_each

where:
there are some predefined constants: the maximum, minimum
and median vertical and horizontal interommatidial
angels, based on the measurements of Seidl.

function max_α(ε, zone):
    return the maximum azimuth angle for each zone,
calculated and interpolated from Seidls measurenments of
the border of the eye

function calc_Δφ_h(ε, α):
    return Δφₕ based on ε and α, calculated from the
predefined constants

function calc_Δφ_v(ε):
    return Δφᵥ based on ε, calculated from the predefined
constants

function calc_offset(Δφₕ, ε):
    return an offset based on Δφₕ and ε, calculated with
some trigonometric functions.
```

The routine was implemented as an R script. The model produces angles for 5440 ommatidia. These are computed once, then stored in a comma separated file for the renderer.

4.1.2 Acceptance Angle

Stürzl et al chose to use an acceptance angle that varies depending on the elevation and azimuth of the ommatidia. Since the interommatidial angles also vary, a static acceptance angle would lead to over sampling in areas of high resolution (e.g. the centre of the eye) and under sampling (at the edge of the eye). The lens diameter also varies between 17 µm and 24 µm over the surface area of the eye, this has been interpreted as an indication for a dynamic acceptance angle by Stürzl et al, however as of now, there aren’t any direct electro-physiological measurements available for the whole eye. The only direct measurements were conducted in the frontal region of the eye and came up with an acceptance angle of 2.6° [24]. Stürzl et al's acceptance angle is not radially symmetric since it depends on horizontal and vertical interommatidial angles. Giger uses a static acceptance angle of 2.6°.

Since there is not enough scientific evidence for a dynamic acceptance angle, the rendering engine in this work uses a static acceptance angle of 2.6°, resulting in a radially symmetric acceptance function, similar to Giger's BEOS application.

4.1.3 Acceptance Function

The Acceptance function is a Gaussian shaped radially symmetric function with a full width at half maximum (FWHM) that is equal to the acceptance angle.

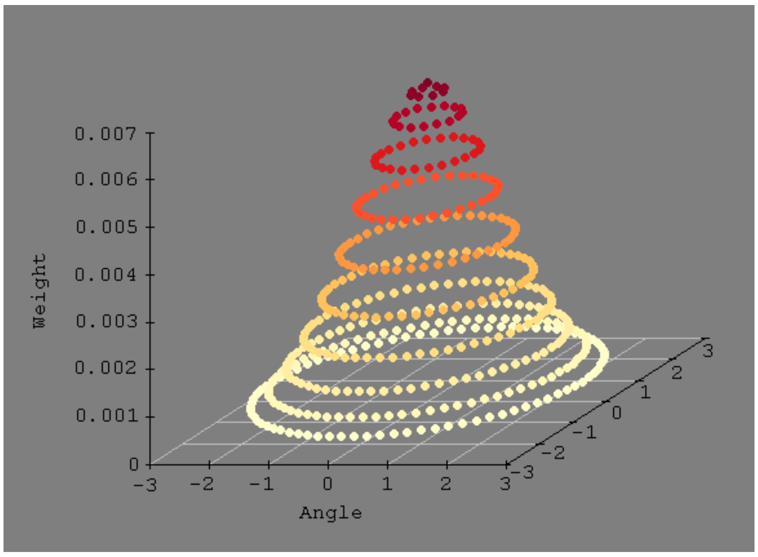


Illustration 4.2: 2D Gaussian sampling function with 462 samples, the weights are shown as a third dimension.

Stürzl et al use 9×9 sampling directions per ommatidium and weight the samples with a Gaussian weight matrix, whereas Giger uses a sampling array of 441 sampling points that are arranged as concentric circles around the optical axis of the ommatidium. Each sample is Gaussian weighted by its distance from the optical axis.

Similarly to Giger a concentric disk sampling method was implemented. This is achieved by creating a square sample matrix with coordinates ranging from -1 to 1 and then mapping the sample points to a disk. Afterwards, the coordinates are normalized to be in the range of -Δρ to +Δρ. By testing different formulas for creating circular samples for various sample sizes, I found that the method described by D. Roșca [25] yields the best results. The formula maps the x, y coordinates of a point in a square to the X, Y coordinates of a point in a disk:

$$(x,y) \mapsto (X,Y) = \begin{cases} \left(\frac{2y}{\sqrt{\pi}} \sin \frac{x\pi}{4y}, \frac{2y}{\sqrt{\pi}} \cos \frac{x\pi}{yx}\right), & \text{if } |x| \leq |y| \\ \left(\frac{2x}{\sqrt{\pi}} \cos \frac{y\pi}{4x}, \frac{2x}{\sqrt{\pi}} \sin \frac{y\pi}{4x}\right), & \text{otherwise} \end{cases}$$

The acceptance function that weights the sample points depending on the distance to the main optical axis of the ommatidium is given by:

$$f(x,y) = \exp\left(-\left(\frac{5\sqrt{x^2+y^2}}{3\Delta\rho}\right)^2\right)$$

The formula approximates a bivariate Gaussian function with FWHM Δρ. For a Δρ of 2.6° this equates to:

$$f(x,y) = e^{(-0.410914(x^2+y^2))}$$

The weights produced from the formula are then normalized to sum up to 1.

Illustration 4.2 shows the resulting acceptance function for 462 samples after the weights were normalized.

4.2 Evaluation

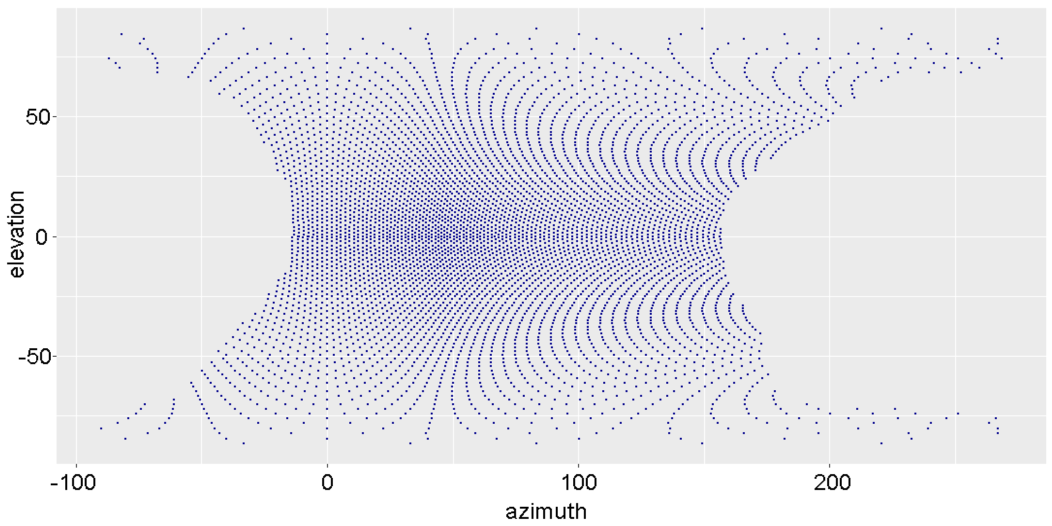


Illustration 4.3: Equirectangular projection of the generated angles for the left eye. The resolution is highest at the equator. The spacing between the ommatidia seem larger for higher elevation angles, however, the effect is exaggerated by the distortions due to the projection. Fig 4.6. shows the ommatidia with a different projection.

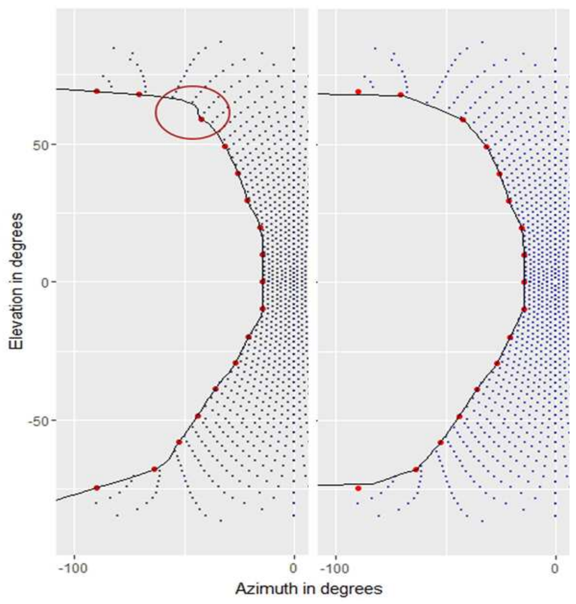


Illustration 4.4: The left portion from fig. 4.3. The red dots are the boundary points measured by Seidl. The black line is the function that interpolates these points. The left graph uses a built-in spline function from R for interpolation (monotone Hermite spline according to the method of Fritsch and Carlson [26]). Stürzl et al also use bicubic interpolation, however, when applied in my model, this produces an unwanted “dent” in the corner of the bee’s eye border (red circle). Other spline interpolation methods (e.g. cubic spline interpolation according to Forsythe, Malcolm and Moler [27]) produced even worse results. Using linear interpolation circumvents this problem and still yields acceptable results (right graph).

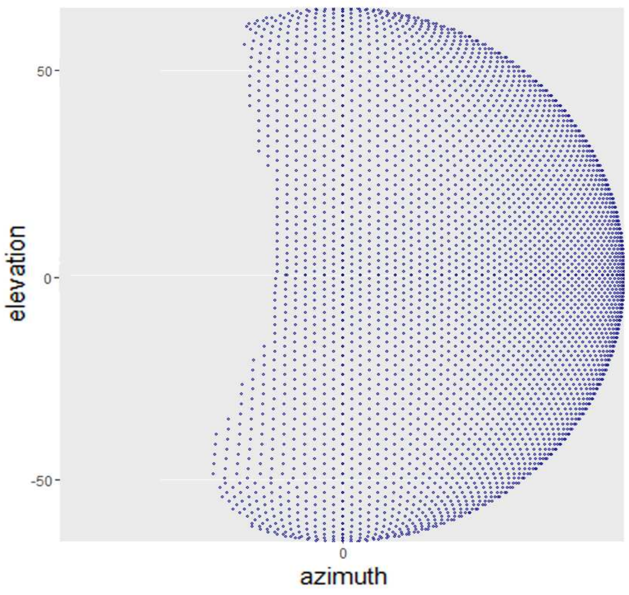


Illustration 4.5: The ommatidia projected onto a sphere as orthographic projection, viewed from the front of the eye. This projection does not introduce as much distortions: one can see that the angles are relatively evenly spaced throughout the eye.

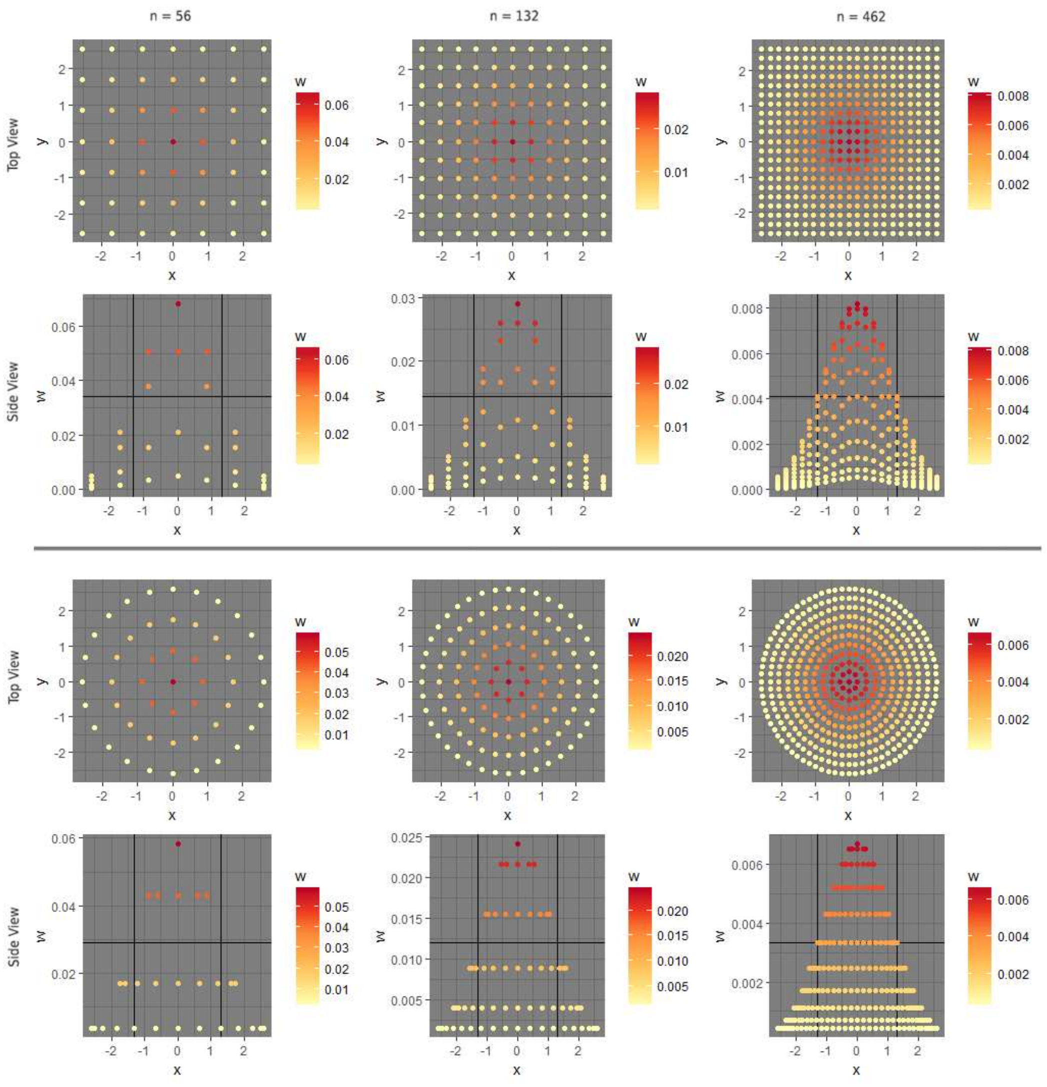


Illustration 4.6: This figure compares the shape of different sampling functions viewed from the side and the top. The side view also visualizes the FWHM (black lines). Stürzl et al use a square sampling function with 81 sampling points, and Giger uses a concentric disk sampling function with 441 sampling points. How the number of samples affect the output image is thoroughly compared in the evaluation part in chapter “Rendering Engine”.

4.3 Visualization of the Ommatidial Array

The model for calculating the ommatidia also creates an ordering, which can be used for assigning x and y coordinates to the ommatidia for representation in 2D space. Starting at the centre of the eye the ommatidia are sequentially numbered in azimuth direction for the x coordinate and in elevation direction for the y coordinate. This leads to a representation where all ommatidia with the same elevation share a row, whereas ommatidia with the same x coordinate can have very different azimuth angles. This produces an image that roughly has the shape of an ellipse, similar to the ellipsoid shaped bee's eye. For creating the other "eye", the eye is simply mirrored.

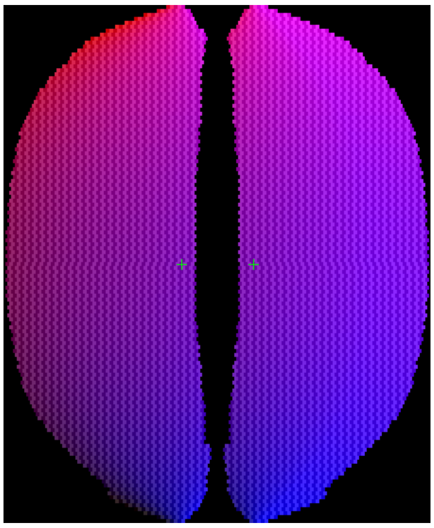


Illustration 4.7: Distribution of ommatidia resulting from the model. The distribution corresponds to the visualization Stürzl et al chose for their bee vision simulation. Each ommatidium is a 4 pixel wide square. To simulate the hexagonal shapes of the facets every second row is shifted by half of the ommatidia's width. The viewing angles of the ommatidia are visualized in the red and blue colour channels: red corresponds to the elevation angle and blue to the azimuth angle. Every second ommatidia is coloured slightly darker so the single ommatidia are distinguishable from another.

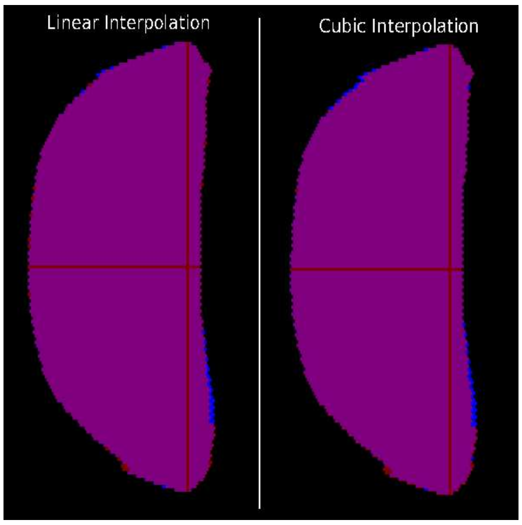


Illustration 4.8: This figure compares the differences between the ommatidial array of Stürzl et al and my representation. Red pixels: my representation, Blue pixels: Stürzl et al's representation, Purple pixels: Overlap between the two representations. The graph also compares which of the eye border interpolation methods is closer to Stürzl et al. As shown, the methods produce similar results, but the overlap is slightly larger when using linear interpolation.

Of course, the visual representation used here is just one visualization method for human interpretation of bee vision. Other representations have been suggested and used. Collins [17] for example uses the elevation and azimuth angles as spherical coordinates on an ellipsoid and simulates the hexagonal facets with a Voronoi diagram.

5 Rendering Engine

The main goal of this work is to provide a similar rendering engine as the one used on www.bee-vision.org, (the latter is based on Stürzl et al's model eye), but with a raycasting technique and without the shortcomings of the online service, as described in the chapter “Related work”.

Therefore I have pursued the following objectives for the rendering engine:

1. The engine should accurately simulate the resolution and field of view of the bee's eye.
2. It should be able to load arbitrary scenes.
3. It should be flexible, i.e. suitable for use in a multitude of different contexts and on different platforms.
4. It should be relatively fast, i.e. able to render several thousand views per hour, thus allowing to render “bee movies”.
5. The source code should be open source, giving other scientists the ability to use and extend the renderer.

At first, I considered to render 360° views and then remap these panoramas to bee views. While simple, this method requires high resolution panoramas for the remapping to be accurate: for a 360×180px panorama, each pixel covers an area of 1° of the field of view. The rendered 360° views therefore should at least have a resolution of 3600×1800px so each pixel covers an area of 0.1°, which results in a sample area of 52×52 pixels for each ommatidium (since the $\Delta\rho$ is 2.6°).

An advantage of this solution is that it is possible to use existing rendering engines such as Blender, so no custom solution has to be developed. Additionally any 360° photograph can be remapped, so no 3D model is required. Disadvantages are that in order to compute the bee view, first an image with 6.48 million pixels has to be rendered, and then remapped again, so the colour values are interpolated multiple times which could lead to inaccuracies. This method also is not very fast. Tests with the Cycles render engine from Blender with 1 bounce and 1 sample per pixel took about 24 seconds just for rendering the 360° panoramas.[2]

This approach was discarded as too slow and too inaccurate, therefore a custom render engine was built that can directly render the bee views.

2 The specifications of the test system can be found in the appendix (1)

5.1 Implementation

The rendering method developed in this thesis is based on raycasting. In the chapter “Computer Graphics Basics” I have explained how raycasting works.

The basic idea for the renderer goes as follows: First the scene is constructed. Then, for simulating the light that an ommatidium receives, the viewing direction, the acceptance angle and the acceptance function of the ommatidium have to be taken into account. For this, for every ommatidium a number of rays are generated. The direction of the rays is calculated from the viewing direction of the bee and the viewing direction of ommatidia, given by $\Delta\varphi_h$ and $\Delta\varphi_v$. The distribution and the number of rays is given by the acceptance function. After the ray intersection, the sampled colours (calculated from the texture of the object) are weighted by the acceptance function, and summed up. The bee view image is then created from the x and y coordinates of the ommatidia.

5.1.1 Choice of Technology

There are multiple approaches that were considered for building the application, ranging from low level GPU programming (e.g. with OpenCL) to high level game engines like Unity. The low level approach was discarded because, even though on paper raycasting seems like an optimal candidate for parallelization, there are several arguments against it: Platform independence is difficult to achieve, GPU code is hard to debug and the learning curve for GPU programming is steep. Even Industry class renderers such as the Corona renderer do not use the GPU [26], since CPU solutions exist that offer acceptable performance.

Nvidia and AMD both offer a raytracing framework for their respective GPUs (Nvidia Optix and AMD Fireray), so there is no need to implement the raycaster from scratch. However, only supporting one vendor is not an option, and porting the renderer to multiple platforms is too complex.

The high level approach was also discarded, because game engines are generally optimized for achieving high frame rates with rasterization, which is a completely different technique than raytracing. While the Unity game engine offers a built-in raycasting function, the function is not meant to be used for rendering, but for collision detection e.g. for bullets. Additionally using a full blown game engine only for the raycasting function is a sort of “overkill”.

Another approach is to use an existing open-source raytracer and extend it with a bee view camera. Candidates that were considered are: OSPRay [27], pbrt [28] Mitsuba [29]. Extracting the functions needed from these raytracers is quite complicated because only a very small subset of the capabilities offered is needed and, as the code is highly optimized, it is often sprinkled with SIMD instructions, which makes it difficult to work with.

The optimal solution is to use a raytracing framework that offers core raytracing functionality such as intersecting rays with the scene, while hiding the underlying acceleration structures and CPU optimizations. Embree, a raytracing kernel developed by Intel, meets these requirements. Additionally it has a good documentation and, even though still under development, the API is stable. Furthermore it is highly optimized for CPUs, achieving good results in benchmarks.

Embree is free and open source, as it is released under the Apache 2.0 license. It runs on all modern x86 and AMD64 CPUs. An advantage of Embree is that it performs the computationally expensive aspects of raytracing in the background (e.g., building the BVH or ray intersection with the scene), while offering a simple interface. [30]

A disadvantage of using Embree is that it is only a raytracing kernel, which means it only offers the minimal functionality needed for calculating the intersection of a ray with the scene. All the other tasks for rendering, like loading the scene, generating the rays, texture mapping or shading have to be implemented separately.

The renderer in this thesis is written in C++. It uses Embree for intersecting the rays with the scene and the Eigen C++ Vector Library [31] for fast vector arithmetic. As these libraries make extensive use of SIMD (Single Instruction Multiple Data), the renderer uses single precision floats.

5.1.2 Overview of the Different Components

The rendering engine has the following main components:

- Scene: The internal 3D model representation.
- Loader: Loads the 3D model into the application from a file.
- Camera: Stores camera settings (e.g. the ommatidium directions and the sampling function), camera to world coordinate transformation.
- Renderer: Renders an image of the scene with the camera.
- Application: Defines the API.

In the following sections, I describe the most important classes belonging to the main components. Code variable names are highlighted in blue, these are referenced later in pseudo code snippets.

Scene

Scene class: This class stores a list of triangle meshes (*meshes*) and additional information such as the scene bounds. On scene creation it initializes Embree by filling its buffers so Embree can build it's BVH, after which no modifications to the scene are possible.

Mesh class: This class has 3 buffers: The *position_buffer* contains the positions of the vertices in the mesh, the *texturecoordinate_buffer* contains the texture coordinates of these vertices and the *triangles* buffer contains triangles. Each triangle holds the indices into the position and texture coordinate buffers of it's vertices *v0*, *v1* and *v2*. The *position_buffer* and *texturecoordinate_buffer* are aligned, meaning that for every vertex *i* the corresponding position and texture coordinate is *texturecoordinate_buffer[i]* and *position_buffer[i].* Every Mesh has a *Texture*.

Texture class: This class is a derived class from *Image.* It stores the texture image data and provides a function *getTextel(P)* that returns the interpolated colour value from the image at position P. For this, either nearest-neighbour or bilinear interpolation is used. Nearest-neighbour interpolation just returns the pixel closest to P, bi-linear interpolation takes into account the 4 closest pixels to P and weights the colour values of these pixels by their distance to P.

Image class: This class is used to store RGB image data. It provides functions for loading data from a file and saving the data to a file. The only file format supported is the portable pixmap (PPM) image format, since it is easy to parse and write. This means that all textures have to be PPM files if the renderer should be able to read them.

Loader

ObjLoader class: This class constructs the Scene from a file. The rendering engine uses the Wavefront Object (.obj) file format, since it is supported by most of the major 3D applications and it is an open, human readable format, so writing a parser is not too difficult. The loader implemented in the renderer is based on the object loader from Intel Embree's tutorials [32].

Object files have the following structure:

```
# the accompanying material library for all objects
mtllib materials.mtl

# the list of vertices of the object
v 493.910614 -77.931282 -14.178329
v 493.032959 -78.255478 -15.455059
v 493.134216 -78.899010 -14.555809
...
```

```
# the list of texture coordinates of the object
vt 0.9930 0.9518
vt 0.9929 0.9511
vt 0.9933 0.9513
...

# define the material name of the object
usemtl texture

# the list of faces of the object
f 1/1 2/2 3/3
f 3/3 4/4 1/1
...

# start next object
v 1308.373169 -452.531982 606.924377
...
```

For every object in the scene there is a list of vertex coordinates, a list of normals (optional), a list of texture coordinates (optional) and a list of faces. The list of faces contains indices into the other lists, defining the vertices of the face:

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
```

These lists have to be parsed and aligned so that they are compatible with the *Scene* format described earlier. All objects are triangulated during the parsing process, because the renderer only supports triangle meshes.

The accompanying material library (*.mtl* file) is loaded when the *mtllib* token is parsed. The material library defines the material properties of the objects. The renderer only supports diffuse textures as material (*map_kd* token), since there is no lighting model implemented.

The *ObjLoader* has an option to convert the coordinates to be right-handed when loading the model, to insure compatibility with applications that follow a different coordinate system convention.

Camera

Camera class: The camera class is used to control the render parameters. It stores the position and direction of the camera in a transformation matrix (*cameraToWorld*). This matrix is used to transform points defined in camera space to world coordinates.

The *camera* class also provides functions for moving, rotating and rolling the camera in all directions.

Illustration 5.1: Images rendered with the pinhole camera. Left: with a FOV of 20°. Centre: with a FOV of 40°. Right: with a FOV of 60°.

PinholeCamera class (derived from *Camera*): This class is used to render images that resemble photographs taken with a normal camera. The class was implemented, since a “human” camera is needed for debugging and verifying that the scene is displayed correctly. The pinhole camera is based on a very simple model and only has the following settings: width and height of the rendered image (in pixels) and horizontal FOV.

PanoramicCamera class (derived from *Camera*): The pinhole camera model only provides a limited field of view (up to 180°, but distortions are already big at 150°), so another camera class is provided that can render images with an arbitrary FOV. This class is useful for investigating the bee’s view, since bees have a FOV that cannot be covered by the pinhole camera. Possible settings for the panoramic camera are: horizontal FOV, vertical FOV, and width of the rendered image.

Illustration 5.2: Image rendered with the PanoramicCamera, covering the full FOV of 360×180°.

BeeEyeCamera class (derived from *Camera*): This class stores the left and right bee’s eyes and the sampler. It is used for rendering the bee views.

The only parameter of the *BeeEyeCamera* is the size of the ommatidia in pixels (default is 4). The other bee’s eye parameters, such as the number of samples per ommatidium and the acceptance angle are controlled by the *Sampler* class. The viewing directions of the ommatidia are controlled by the *BeeEye* class.

BeeEye class: This class stores an array of *Ommatidiums*. It constructs the ommatidia from a *.csv* file which contains the elevation and azimuth angles (defining the viewing direction) for each ommatidium. The file was created with the R script described in chapter "Bee's Eye Model". During the loading process x and y coordinates (used for visualizing the ommatidial array) are calculated. The coordinates describe the position of the ommatidium on the rendered image.

Sampler class: This class simulates the acceptance function of the bee's eye. Based on the acceptance angle and the number of samples (*num_samples*), it creates an array of 2D sample points (*samples*), each point describing the angular distance from the centre of an ommatidium. It also calculates an array of corresponding Gaussian weights (*weights*). It can either produce a square sampling distribution or concentric disk sampling distribution. Since a static acceptance angle is used for all ommatidia, the sample points only need to be calculated once.

Renderer

Renderer class: This class renders an image from the scene based on the active camera's parameters. In the following section, code snippets are used to illustrate the rendering process. The complete pseudo code can be found in the appendix.

For rendering the image, first the rays have to be constructed. In case of the *PinholeCamera* and the *PanoramicCamera* for every pixel in the image the ray direction is calculated from the camera's parameters. In case of the *BeeEyeCamera* for every ommatidium *Sampler.num_samples* rays are constructed.

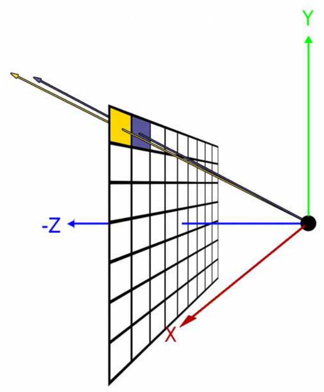


Illustration 5.3: PinholeCamera ray construction: The rays are shot through an imaginary plane placed 1 unit away from the origin. The camera plane is calculated from the width, height and FOV settings. The ray direction can then be calculated with trigonometric functions.

PanoramicCamera ray construction: Looping through the pixels of the image, the rays are rotated by the horizontal angular spacing (calculated by dividing the horizontal FOV by the width of the image) and the vertical angular spacing (calculated by dividing the vertical FOV by the height of the image).

BeeEyeCamera ray construction: For every ommatidium the main ray direction is given by the elevation and azimuth angles. To calculate the ray direction of each sample of the ommatidium, the deviation to the main angle (given by the sample point) is added to the main ray direction. The angular coordinates are then transformed to Cartesian coordinates:

```
dir = toCartesian(ommatidium.azimuth + sample.x,
          ommatidium.elevation + sample.y)
```

The rays are calculated in “camera space”: the default camera position is the origin of the coordinate system, looking down the -Z axis. The ray direction is then multiplied with the linear part of the *cameraToWorld* matrix, to rotate the rays in accordance with the camera’s orientation. The ray origin equals the camera’s position (stored in the translation part of the camera matrix):

```
ray.dir = camera.cameraToWorld.linear() * dir
ray.origin = camera.cameraToWorld.translation()
```

After construction, the rays are intersected with the scene by using the Embree *intersect* function. This function returns the object ID (*hitpoint.objID*) of the first object that is hit, the triangle Id (*hitpoint.triID*), and the barycentric u, v coordinates (*hitpoint.u*, *hitpoint.u*) of the hit point.

```
hitpoint = Embree.intersect(scene, ray)
```

Barycentric coordinates describe the 2D position of a point *P* in relation to a triangle. This is done with a linear combination of the triangle’s points (*P0, P1, P2*):

$$P = wP0 + uP1 + vP2$$

$$P = (1 - u - v)P0 + uP1 + vP2$$

$$\text{since: } u + v + w = 1$$

So given a hitpoint on a triangle one can calculate its texture coordinate:

```
mesh = scene.meshes[hitpoint.objID]
triangle = mesh.triangles[hitpoint.triID]
```

```
vt0 = mesh.texture_coordinates[triangle.v0]
vt1 = mesh.texture_coordinates[triangle.v1]
vt2 = mesh.texture_coordinates[triangle.v2]

pt = (1 - hitpoint.u - hitpoint.v)*vt0 +
        hitpoint.u*vt1 + hitpoint.v*vt2
```

To compute the colour, the texture coordinate *pt* is bi-linearly interpolated by the *getTexel* function:

```
color = mesh.texture.getTexel(pt)
```

In case of the *BeeEyeCamera* all sampled colours of an ommatidium are weighted by *Sampler.weights* and summed up. The pixels are then expanded to *ommatidium_size* and arranged with the x and y coordinates of the ommatidium. The two eyes of the bee are rendered separately and then combined.

There is also a function that does not render a 2D image, but returns three arrays containing the elevation, the azimuth and the colour data for each ommatidium. This function can be used for neural agents, or for constructing other visual representations of the eye.

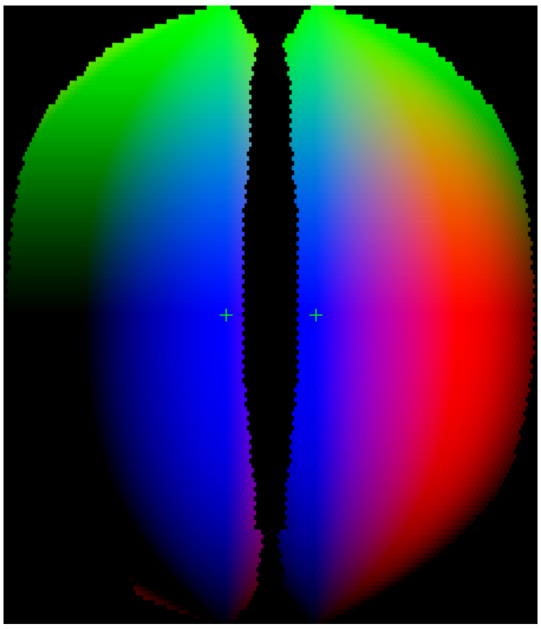


Illustration 5.4: This image shows the 3D Cartesian coordinates of the ommatidia's direction vectors, calculated from the elevation and azimuth angles. The red channel is the x coordinate, the green channel is the y coordinate and the blue channel is the z coordinate of the direction vector. The green cross is the viewing direction of the bee.

Application

BeeViewApplication class: This class simplifies the process for rendering an image, by providing a simple API for controlling all of the renderer's settings. On construction of the class, the scene is loaded, the renderer is initialized, and the cameras are set up. The camera can be controlled through different functions, the camera parameters can be set, and the render mode can be changed (bee mode, panoramic mode or pinhole mode). The API only exposes primitive data types, so that it can easily be interfaced with from other programs.

The C++ code is best built as a shared library, and used by including the *beeview_api.h* header file. The source code can be found on Github [33]

5.2 Evaluation

In this section I will go into how different rendering parameters effect the output and the performance of the renderer.

The performance of raycasting directly depends on the number of rays to cast and the amount of polygons in the scene. In case of this renderer these are given by the bee's eye model and the scene described in chapter "3D Environment". The amount of rays needed for rendering a bee view is $2N_oN_s$. Where N_o is the number of ommatidia and N_s is the number of samples per ommatidium. So for 462 samples per ommatidium the renderer has to generate 5 026 560 rays and perform as many intersection tests. The scene has over 1 million faces that need to be tested for intersection. The amount of rays needed to render an image equals the total amount of samples, as each sample produces one ray.

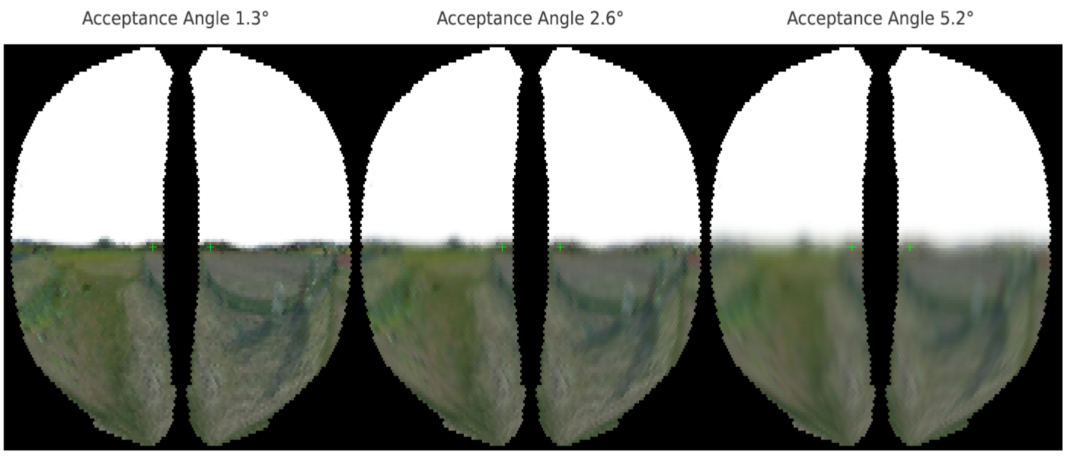


Illustration 5.5: The effect of the acceptance angle on the output of the image. The larger the acceptance angle, the blurrier the image. Also, objects that are further away are blurrier than closer objects, since with greater distance the acceptance angle covers a larger area. One can conclude that bees are short-sighted.

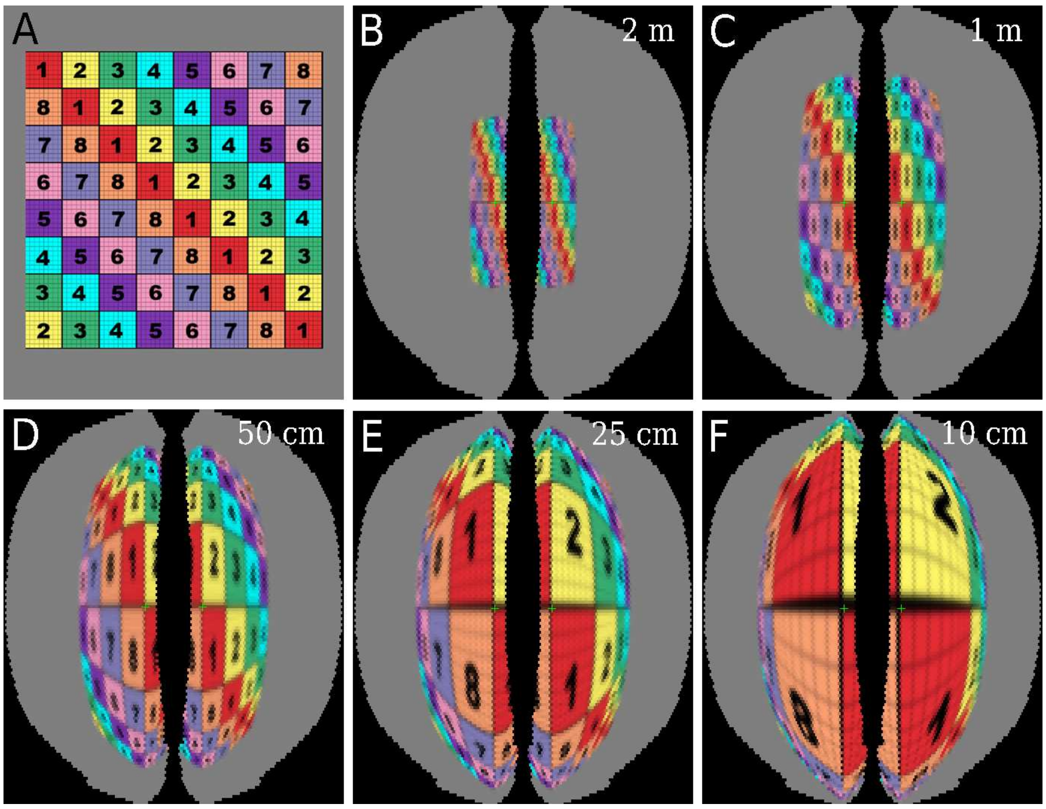


Illustration 5.6: These renderings show the simulation of how a bee sees a 2×2m image from different distances. Image A: The test scene, rendered with the pinhole camera. The test image shows squares of different colours with numbers in them. Each square has a width and height of 25 cm. Because of the square form, distortions are easily identified. The numbers and colours help distinguish the different squares. As mentioned before, the renderer only simulates the resolution and not the colour perception of bees, so the colours are purely for distinguishing the squares. Image B: The test image seen from a distance of 2 m. Image C: 1 m. D: 50 cm. E: 25 cm. F: 10 cm. The test series shows that objects in the centre of the eye are enlarged, since the resolution of the eye is highest here. Also, the closer the object, the bigger are the distortions, because the object covers a larger part of the field of view. Additionally one can see that only a small portion of the field of view of the eyes overlap, an indication that bees do not rely on stereo vision for reconstructing the 3D shape of objects. The settings for the bee camera used: acceptance angle 2.6°, disk sampling, 132 samples per ommatidium.

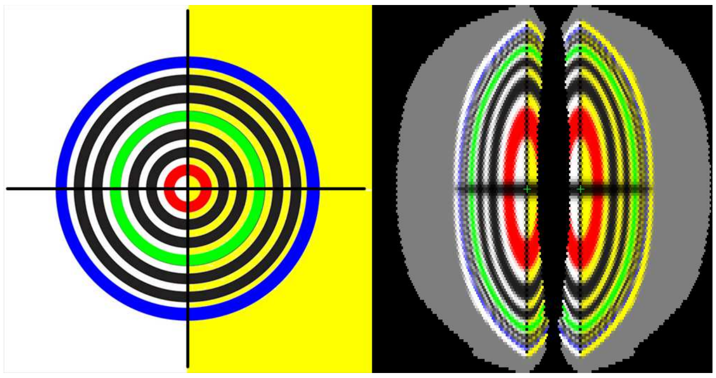


Illustration 5.8: The left image shows the test image. It is placed 25 cm in front of the bee as a 2×2m canvas. The concentric circles have the same width, and are evenly spaced. The right image shows the rendered bee view. One can see that the inner circles are larger than the outer circles and that the line at equator of the eye is thicker than the vertical line. Also the spacing between the circles becomes smaller at the edge of the picture.

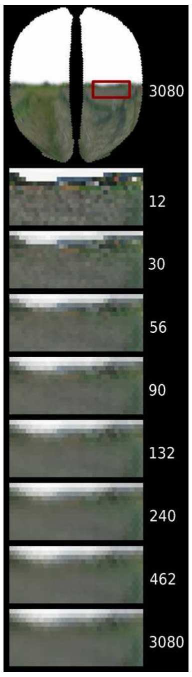


Illustration 5.9: These renderings show a portion of the bee view (the red rectangle) for different sample sizes. The following is a comparison of how the number of sample points affects the simulated bee vision. Since in reality there are a lot of light rays that reach the eye, more rays in the simulation lead to an image that is closer to reality[3]. The number of sample points is a big factor in the performance of the renderer. Therefore the goal is to minimize the number of samples needed, but that the output of the image is not affected too much by the reduction of rays. For this a series of bee views was rendered, with the number of sample points per ommatidium ranging from 12 to 9900. The different numbers of sample points (n) used can be found in the appendix. At the beginning (until n = 240) the differences between the bee views are quite large. From n = 240 to n = 462 the image still becomes smoother, but the differences are almost not noticeable. From n = 462 there is no conceivable difference between the rendered bee views.

In summary, it can be stated that at least a sample size of 56 should be used if performance is very important. A good trade-off between performance and accuracy is 240 sample points per ommatidium, and a sampling size of 462 should be used to get the most accurate results.

Number of samples	Number of Rays	Average time in ms	Rays per Second
56	609 280	1053	578 613
240	2 611 200	3635	718 349
462	5 026 560	6507	772 485

Table 1: Time it takes to render an image depending on the number of samples per ommatidium. On average, the renderer achieves 689 816 rays per second. Since for casting the rays Embree utilizes all cores of a system, running the renderer on a desktop processor with multiple cores should significantly speed up the process.

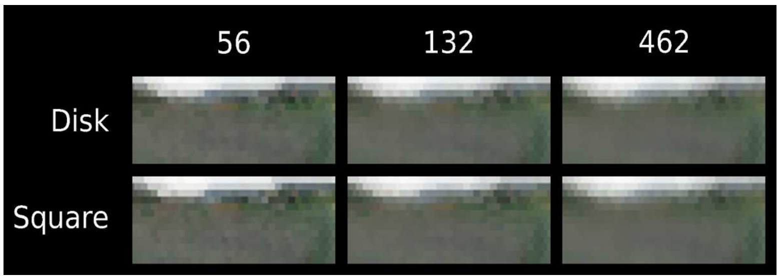


Illustration 5.10: Comparison of differences between using a square sampling distribution (similar to Stürzl et al [12]) and a concentric disk distribution (similar to Giger [13]). There is almost no difference between the two methods, except for small sample sizes, as square sampling covers a larger area (2Δρ = width of the square = diameter of the disk). However for larger sample sizes the differences are not conceivable any more, since the samples at the edge have a small weight.

3 The amount of light a receptor receives can not be quantified by rays, since rays are just an idealized model of light, but rather by the amount of photons or energy transported by the light.

6 Python Interface and Radar-Track Movie-Maker

In this chapter, an extension for the Radar Track Analysis tool from the *FU Biorobotics Lab* is presented. The Radar Track Analysis tool is a Python module that provides interpolation, plotting, and analysis functionality for insect flights that were tracked with radar at the testing site in Hesse described in the chapter "3D Environment". The module is used to study various aspects of bee navigation, for example it was used to investigate the effects of neonicotinoid based insecticides on bee navigation [34].

The flight data was gathered with harmonic radar, using a small transponder (11mm height) fixed on the thorax of bee. The radar has a sampling rate of 3-5 seconds and a range of about 900m. The flight data is available as 2D x, y coordinates that describe the distance in meters from the radar and a time stamp. There can be larger gaps in the flight data, caused by obstacles between the radar and the bee or if the bee flies too low. The data is interpolated and smoothed by the tool to a frequency of 1s. The flights can be plot onto a map that is a high resolution orthophoto of the area. The module can also automatically distinguish between homing and search flights by measuring the curvature of the tracked flight.

The tool can also render 2D bee views that display a small portion of the map surrounding the bee's position, rotated so that it aligns with the bee's flying direction (compare figure 6.2 for example plots).

The goal of this part of the thesis is to extend the Radar Track Python module to include the 3D environment and to render "bee movies" with the rendering engine based on the tracked flights. The following main objectives for the extension can be defined:

- Developing an interface for the C++ rendering engine, so that it can be used from within the Python module.
- Implementing functions for rendering, saving, loading and displaying the flights.

6.1 Implementation

This section describes how the objectives were realized. The "interface" part describes the interface for the C++ rendering engine, the "MovieMaker" part describes the extension of the Radar Track module.

6.1.1 Interface

Every time the renderer is started, the 3D scene has to be loaded from the Object file which takes a substantial amount of time (about 12 seconds)[4]. Therefore the communication with the renderer should either be done at run time, or all the information needed to render the images must be provided beforehand. The following options for developing an interface to the C++ renderer were evaluated:

- File input/output: The data needed to render images (such as the camera's orientation and other settings) can be read from files. This method is quite inflexible, as all positions and directions of the camera have to be known beforehand. In case of the radar tracks this is true, but for example a neural agent communicating with the renderer dynamically updates its position depending on previous views. Additionally, file parsing is relatively slow compared to other methods.
- Client-Server model: communicating over a protocol via REST, SOAP or RPC, or defining a custom messaging system via 0mq and Google Protobuf.
- Providing bindings to the C++ code with Python using SWIG, pyrex, ctypes or Cython.

Wrapping the C++ code with Python was chosen, because this is the most direct way to communicate with the renderer. with this method, no protocols have to be defined, no sockets have to be used, and the C++ application does not have to be started separately, but can be used from within python. Also Python is widely used in scientific programming, so this provides an interface that can easily be used with other scientific applications.

Cython was deemed best suited for wrapping the C++ code. Since only the API defined in *beeview_api.h* has to be wrapped there is no need of a semi automatic wrapping tool like SWIG. Cython is a language that adds C data types to the python language so that C functions can be called from within Cython. It is a superset of Python, therefore any valid Python code is also valid Cython code. It is a compiled language that creates C code from the Cython code and compiles the code into a Python extension module that can be used with normal Python. [35]

For every C++ function that should be wrapped, a corresponding Cython function was written that calls the C++ function. The result is the *beeview* python package that, after being built with the Cython compiler, can easily be imported into any Python module.

4 The specifications of the test system can be found in the appendix (1)

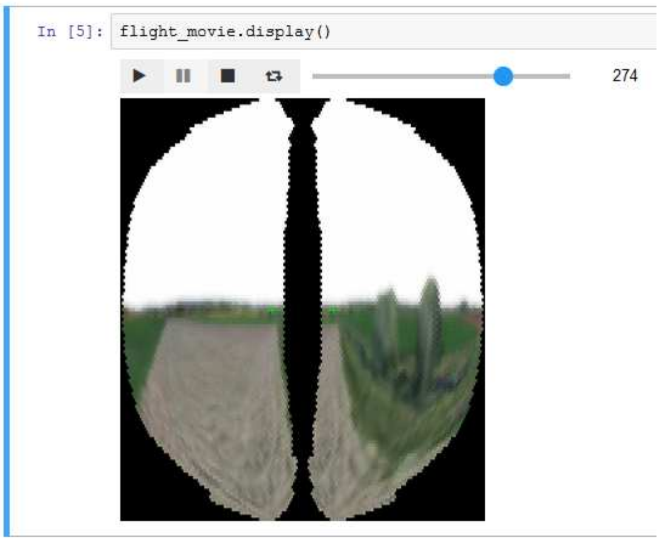
The *beeview* package defines a *Renderer* class that provides all the functionality of the C++ rendering engine. Also a method for measuring the distance from a point to the next object in a given direction was implemented. This method is needed to calculate and adjust the camera height above ground.

6.1.2 Movie-Maker

In this section the implementation details for extending the Radar Track module are provided.

To be compatible with the Radar Track module the coordinates of the 3D model have to be in meters. Additionally the model has to be oriented so that the Z coordinate points north. Also the position of the radar in the 3D model has to be known, so the coordinates can be converted to be relative to the radar.

The Radar Track tool has a *Flight* class, that contains the interpolated positions and time data of the tracked bee flight. The 3D model and the interommatidial angles (that were calculated from the model eye) were integrated into the data module of the Radar Track tool, so that they do not have to be provided separately which simplifies the process of rendering a bee view.



```
In [5]: flight_movie.display()
```

Illustration 6.1: The Radar Track module is meant to be used interactively with Jupyter Notebook (IPython). This image is a screenshot of the IPython Widget that is used for displaying the flight movies. The widget provides a slider and play/pause functionality, so one can simply browse through the frames in the FlightMovie.

The *FlightMovie* class contains the rendered flight as a list of numpy arrays that contain the image data. It also stores the *Flight* object that was rendered and the settings that were used to render the flight. It has a function for saving the *FlightMovie* to disk. For this different options are possible: The movie can either be saved as a multi-page TIFF, or as a folder containing a sequence of jpeg images, or as a mp4 movie. An accompanying JSON file is created when saving the movie, containing information so that the *FlightMovie* object can be loaded from these files.

The *MovieMaker* class is used for rendering the flights. On construction of the class, the *Renderer* of the *beeview* package is initialized. For rendering the flight, a function is provided that takes a *Flight* object and returns a *FlightMovie* object. This render function converts all the positions in the *Flight* to be compatible with the 3D coordinates of the 3D model. Using the *Renderer.get_distance* function the camera's position is calculated, so that the camera is 5m above the ground (the flight height of bees is about 3-7 meters). "Ground" is the object under the camera with the smallest y coordinate (as y is the "up" coordinate). Gaps in the tracked flight are linearly interpolated. The direction of the camera is calculated from the flying direction of the bee. For each position a bee view is rendered and stored in the *FlightMovie* class.

6.2 Evaluation

The working python code examples in this section give an overview of the *beeview* python package and the extension of the radartrack module. The code snippets show how the python package simplifies the use of the renderer by providing a clean and simple API. In conjunction with IPython Notebook one can use the renderer interactively.

Beeview Python package

This code snippet shows some of the features of the renderer and how the rendered bee views can be processed. A list of the functions of the *beeview* package is available in the appendix 3.

```python
import beeview
from PIL import Image
import matplotlib.pyplot as plt

renderer = beeview.Renderer('scene.obj', 'ommatidia.csv')

renderer.position = [0, 0, 1]
renderer.direction = [1.5, 0, -1]
img = renderer.render()

# plot the image with matplotlib
plt.imshow(img)

# save the image with PIL
img = Image.fromarray(img)
img.save('beeview.png')

# change the camera mode
renderer.mode = beeview.Renderer.Pinhole
renderer.set_pinhole_fov(80)
img2 = renderer.render()
```

Radar Track module extention

The following code snippet shows how to render a flight with the Radar Track module. Additionally the saving, loading and displaying of the flight is examplified.

```
import radartrack as rt
import beeview

# automatically loads the scene from rt.data
movie_maker = rt.movie.MovieMaker()

# get a flight
flight = rt.data.flights['2013', ...][0]

# render the movie, automatically adjusts the camera
height for every position
flight_movie = movie_maker.render(flight, 'flight_name',
height = 5)

# save as .tiff, image sequence and .mp4 files
flight_movie.save(_as = ['tiff', 'mp4', 'sequence'])


# display the flightmovie in IPython Notebook
flight_movie.display()

# render settings can be adjusted via the renderer
movie_maker.renderer.set_ommatidium_size(8)

# pre-rendered flights can be loaded from disk
flight_movie = rt.movie.load('flight_name')
```

The temporal resolution of the movies is constrained by the temporal interpolation of flight tracks, which are only interpolated for each second. This means a 25 minute flight yields a movie with 25 * 60 (1500) frames. Playing this movie with a smooth 21 frames per second, results in a clip of 71 seconds.

In 2013, 153 flights were tracked, with a total tracking duration of 31 hours and 24 seconds (111624 seconds). Rendering all flights with 462 samples per ommatidium, would take 111624 * 6.5 seconds, which is about 8 days[5]. Using only 56 samples per ommatidium, it would still take 1 day and 10 hours. Therefore the loading functionality is important so that the flights can be

5 The specifications of the test system can be found in the appendix 1. Running the renderer on a faster computer with more cores would reduce the amount of time significantly, as doubling the amount of cores should roughly cut the time in half.

rendered beforehand and saved to disk. Then the flight movies can be loaded when needed.

As mentioned before, some flights have gaps due to lost radar signal. There is no positional information available for the gaps, only the duration of the gap is known. The gaps are linearly interpolated, which means the bee is simulated to fly straight from the starting point of the gap to the end point of the gap. This has the effect that at the transitions to and from the gap there can be an abrupt change in flying direction.

An example movie of a rendered flight can be found online:

https://youtu.be/e6fLh5XhvIg

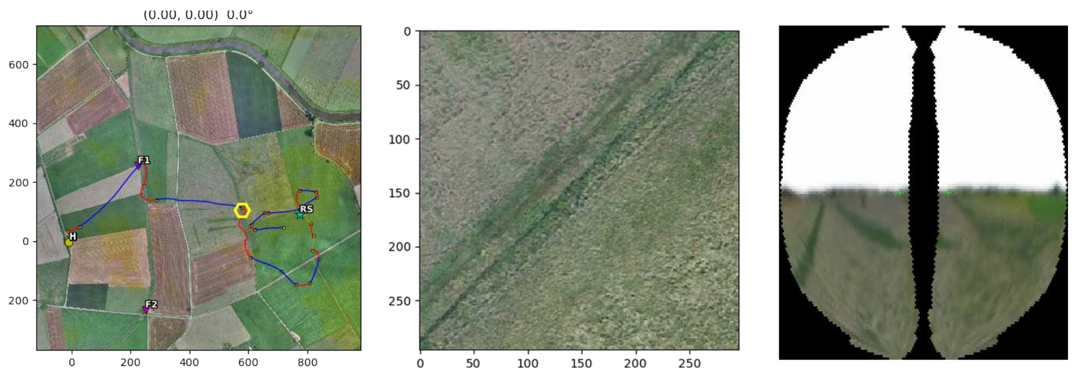


Illustration 6.2: Left image: an example plot of a flight from the release site (RS) to the hive (H). Red is search flight, blue is linear flight and the dotted yellow lines are gaps. The yellow hexagon is the position of the bee. The central image shows the 2D bee view from that position (a 25×25m portion of the map), rendered with the Radar Track module. The right image shows the bee view rendered with the help of the *beeview* python package.

7 Discussion and Outlook

7.1 Summary

The objective of this thesis was to simulate bee vision. 4 major tasks were accomplished:

- Extending a 3D model of a previously mapped area in Hesse to include its surroundings
- Implementing a model of the bee eye's optic apparatus
- Building a renderer that uses the 3D environment and the bee's eye visual model to simulate bee views
- Building an application that can render movies from bee flights

3D environment. Visual landmarks in the horizon panorama of a scene may be crucial for insect navigation. Therefore, the existing 3D model had to be extended to include a horizon panorama. After evaluating different methods a solution was found, by which the surroundings were made visible from within the model. For this a panoramic image of the surroundings was prepared by removing all objects from the panorama that are also contained in the 3D model. The panorama is used as a skydome for the model. The skydome is oriented in such a way, that it accurately represents the reality. To evaluate the accuracy of the method, different viewpoints were compared from within the 3D model and from the real scene. Depending on the viewpoint, deviations from the real scene were found, however they all spread out in an acceptable range.

Bee's eye Model. A bee's eye model that represents the spatial resolution and field of view of a honeybee's eye was implemented. The model is based on microscopical and electrophysiological measurements [11], [24]. For this, an R script was implemented that produces interommatidial angles of the bee's eye in accordance to a model of Stürzl et al [12]. The model eye used in this thesis uses a static acceptance angle of 2.6° for all ommatidia. The acceptance function is modulated by a Gaussian weighted, concentric disk sampling function.

Renderer. Using the bee's eye model, a renderer based on raycasting was implemented. The renderer can load arbitrary scenes from *.obj* files. Written in C++ and utilizing Intel's high performance raytracing kernel library Embree it is efficient. While not achieving real time rendering, the renderer can be used interactively, since it can render a bee view in under one second (depending on the settings and the system it is run on).

The renderer is not limited to rendering bee views, but can also render normal images from a pinhole or a panoramic camera. It has a simple API so it can easily be interfaced with from other applications. The *beeview* Python package provides bindings to the C++ functions of the renderer. The renderer is implemented in a way that all the settings of the renderer are flexible. This means that the renderer can easily be used with different interommatidial and acceptance angles for simulating the vision of other insects.

The optimal settings for rendering bee views were determined so the performance is maximized while still maintaining accuracy.

MovieMaker. An existing Python package for analysing radar tracks of bee flights was extended so that it uses the 3D model and the renderer for creating movies from the tracked flights. To render the flight movies, the 2D radar coordinates have to be transformed to 3D coordinates and gaps in the radar data have to be handled. Additional functions were implemented for saving the flight movies in different formats, for loading the movies from disk, and for displaying the movies in IPython Notebook.

7.2 Application

The enhanced 3D model is an ideal environment for an artificial neural agent: it covers everything one can see, and as it is based on a real environment it is highly realistic. The C++ API and the *beeview* Python package provide all the functions needed for a neural agent like the one implemented by Müller [5]:

- Functions for moving, rotating and rolling the camera, so all the possible movements of a bee are covered. Furthermore it is possible to set the camera direction directly, or via a *look_at(point)* function.
- A render function that returns the elevation angles, the azimuth angles and the sampled colours of all ommatidia as continuous arrays for visual input of the agent.
- A function for measuring the distance from a point to the next object in a specific direction, that can be used for measuring the height above ground and setting the camera's position accordingly.

The flight movies rendered by the MovieMaker can be used to study the optical flow seen by the bees. Furthermore, the flight movies can be used to explore how features such as landmarks or ground structures play a role in the navigation of bees.

The renderer can also be used for evaluating pattern and shape recognition experiments, as in [13], [36], [37], by placing test images at a specific distance from the camera.

Additionally the renderer can be used for educational purposes to demonstrate how different parameters of the compound eye affect insect vision.

7.3 Improvements

There are multiple aspects that can still be improved.

Loading the high resolution textures of the model has the biggest impact on the start-up time of the renderer (about 10s). Using a faster image library instead of the simple ppm loader could speed up the process. Only supporting the ppm file format for images is not optimal, but can easily be improved by using a different image library.

The way gaps in the radar data are handled, when rendering flight movies, may result in abrupt viewing direction changes at the transition points of the gap. This can be improved by implementing a better interpolation method that takes into account previous flying directions.

For very close (cm range) objects the bee's eye model does not yield correct views, since it assumes that both eyes are at the same position. However this is not a problem most of the time, since bee's eyes are very close together.

The simulation is of course just a simplified model of reality. The model eye for example only takes into account the spatial resolution of honeybee's eyes. The model could be extended to include the light intensity received by the ommatidia and the spectral sensitivity of the ommatidia. For this the 3D environment has to include UV information. This could be done by recording the scene with a camera sensitive to UV-light and storing the recorded UV-data in the red channel or alpha channel of the texture. Additionally only the compound eyes were modelled, but for a complete bee vision simulation the ocelli should also be taken into account.

The 3D model could be extended by including light sources and material properties. Based on these the renderer could render shadows and other light effects by tracing the rays for multiple bounces. The polarization of light could be simulated, as some ommatidia of the bee's eye are sensitive to it.

The scene could be refined by using a subdivision mesh (Embree is capable of handling subdivision meshes) [38].

In the summer of 2017 the *Biorobotics Lab* recorded a larger 3D model. Embedding it in a DEM would be a good method for expanding the model, since the new model covers all scene objects that are near the testing area, and the elevation data is sufficient for modelling the far away hills. The method followed in this thesis (with one panorama taken from the centre of the model) wouldn't yield good results, because the deviations of the horizon, depending on viewpoint would be larger, as the model is larger.

7.4 Conclusion

In this thesis the conceptualization, implementation, evaluation and application of a bee vision simulation was presented.

For this, an existing 3D model was refined and a renderer was developed that can generate bee views from this – or any other – model.

The renderer offers multiple advantages:

- It is well documented and open source. [33]
- It is easy to use through a Python API.
- It is fast, as it is written in C++ and utilizes the Embree raytracing kernel.
- It is flexible: any scene can be loaded, different rendering modes and camera settings are possible.
- It is based on an accurate bee's eye model, constructed from scientific evidence.

In this thesis the renderer is applied for rendering movies from bee flights tracked with radar. For this the renderer is an ideal tool, but it can be used for multiple other scenarios, e.g. as input for an artificial neural agent, for educational purposes or for further exploring insect navigation.

Acknowledgements

I would like to express my gratitude to my thesis supervisor Tim Landgraf for giving me the freedom to write this thesis "abroad" so I can be with my girlfriend. Also I would like to thank him for being there when I needed support whilst giving me the space I needed.

I am also grateful to the members of the Neorocopter group of the FU Biorobotics Lab for giving me the opportunity to take part in the radar experiments conducted in Großseelheim. Special thank goes to Julian Petrasch for recording the 3D Model used in this thesis.

I would also like to thank everyone who helped with proofreading and giving feedback especially my father for the long hours spent.

Thank you, Laura, for your moral support, for enduring me in stressful times and for feeding me when I forgot to eat.

8 Appendix

8.1 Test System Specifications

All benchmark tests were carried out on a laptop from 2012 with a 2.9GHz i7-3520M dual-core-processor and 16 GB RAM.

8.2 Sample sizes

The following sample sizes for the concentric disk sample formula yield an evenly distributed, perfectly round distribution. The formula for finding “good” sample sizes is: x * x + x for all odd x (x ϵ ℕ x > 2). Other sample sizes result in a distribution where the outer concentric ring is not complete.

2	462	1722	3782	6642
12	552	1892	4032	6972
30	650	2070	4290	7310
56	756	2256	4556	7656
90	870	2450	4830	8010
132	992	2652	5112	8372
182	1122	2862	5402	8742
240	1260	3080	5700	9120
306	1406	3306	6006	9506
380	1560	3540	6320	9900

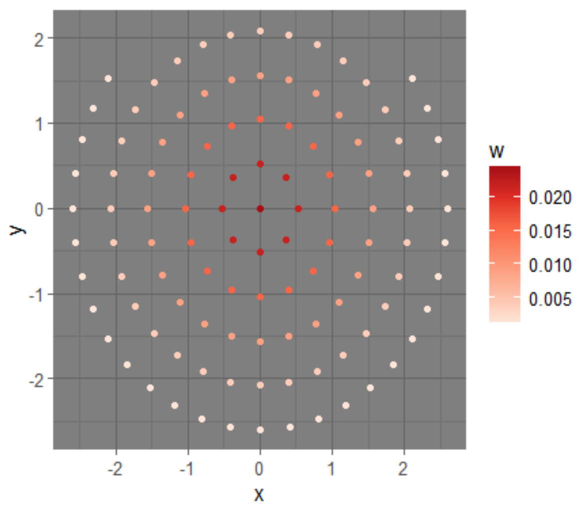


Illustration 8.1: Image of the concentric disk mapping if used with a number of sample points that is not in the list.

8.3 Rendering Pseudo Code

The following pseudo code illustrates the simplified process of rendering a bee view image. The code is explained in detail in the implementation section of the “Renderer” chapter.

```
foreach ommatidium in beeEye.ommatidia:
    color = 0
    foreach sample in sampler.samples:

        # calculate the viewing direction
        dir = toCartesian(ommatidium.azimuth + sample.x,
            ommatidium.elevation + sample.y)

        # setup the ray
        ray.dir = dir * camera.cameraToWorld.linear()
        ray.origin = camera.cameraToWorld.translation()

        # intersect the ray with the scene
        hitpoint = Embree.intersect(scene, ray)

        # get the texture coordinates
        mesh = scene.meshes[hitpoint.objID]

        triangle = mesh.triangles[hitpoint.triID]

        vt0 = mesh.texture_coordinates[triangle.v0]
        vt1 = mesh.texture_coordinates[triangle.v1]
        vt2 = mesh.texture_coordinates[triangle.v2]

        # calculate the hitpoint position on the texture
        pt = (1 - hitpoint.u - hitpoint.v)*vt0 +
                hitpoint.u*vt1 + hitpoint.v*vt2

        # get the color from the texture
        color = mesh.texture.getTexel(pt)

        # weight the color with the sampler
        weight = sampler.weights[sample]

        # add to color of ommatidium
        color = color + w * sampled_color

    end_foreach

    # color the pixel in the image
    image[ommatidium.x, ommatidium.y] = color

end_foreach
```

8.4 Beeview API functions

The following list of functions gives an overview of the capabilities of the *beeview* Python package. The complete source code and documentation thereof can be found on Github [33].

class beeview.**Renderer**

This class wraps the C++ BeeViewApplication class.
It provides all the functions necessary for rendering Images from a loaded scene.

__init__()

Initialize the Renderer.

Parameters: **- scene** (*string*) The location of the .obj file containing the scene.
- ommatidia (*string*) The location of the .csv file containing the interommatidial angles.

Returns: *self*

mode : *int*

Current render mode. 0 : Renderer.BeeEye, 1 Renderer.Panoramic, 2 : Renderer.Pinhole

position : *sequence of floats*

The position of the active camera.
x,y,z coordinates

direction : *sequence of floats*

The direction vector of the active camera.
x,y,z coordinate. Auto normalized.

get_image_size()

Get the rendered Image Dimensions (width, height)

Returns: Width, height of rendered Image in pixels.
Return type: *tuple of ints*

get_scene_bounds()

Returns the axis aligned bounding box of the scene.

Returns: The upper x,y,z coordinates and the lower x,y,z coordinates of the bounding box.
Return type: *list of tuples*

get_settings()

Get all render settings as Dictionary.

Returns: Dictionary of render settings.

measure_distance()

Measures distance to next object in direction of dir.

Parameters: - **pos** (*list of float*) the x, y, z coordinates of position.
- **dir** (*list of float*) the x, y, z coordinates of dir vector. Will be normalized by the function.

Returns: The distance from pos to the next object in direction of dir. If no object in that direction return -1.

Return type: *float*

render()

Renders an image with the current camera.

Parameters: - **agent** (*bool*) defines if the output should be optimised for a neural agent. If true the returntype changes. Default is false.

Returns: **If agent is false:** Array of shape: [height, width, 3]. Color data is uint8 with range 0-255.
If agent is true:
If BeeEye mode: Dictionary with keys: “azimuth”, “elevation” and “color”. Azimuth contains a list of azimuth angles, Elevation contains list of elevation angles and color a list of lists containing the color values for each ommatidium.
If Panoramic or Pinhole mode: Array of shape [width*height,3] containing the color data

Return type: *array_like*

roll_left()

Roll the active camera left.

Parameters: **- degrees** (*float*)

roll_right()

Roll the active camera right.

Parameters: **- degrees** (*float*)

rotate_down()

Rotate the active camera down.

Parameters: **- degrees** (*float*)

rotate_left()

Rotate the active camera left.

Parameters: **- degrees** (*float*)

rotate_right()

Rotate the active camera right.

Parameters: **- degrees** (*float*)

rotate_up()

Rotate the active camera up.

Parameters: **- degrees** (*float*)

set_acceptance_angle()

Set the acceptance angle for the bee eye camera. Default is 2.6°.

Parameters: **- acceptance_angle** (*float*)

set_num_samples()

Set the number of samples per ommatidium for the bee eye camera. Default is 132.

Parameters: **- num_samples** (*int*)

set_ommatidium_size()

Set the ommatidium size for the bee eye camera. Default is 4.

Parameters: **- ommatidium_size** (*int*)

set_panoramic_hfov()

Set the horizontal field of view for the panoramic camera.

Parameters: **- h_fov** (*float*)

set_panoramic_vfov()

Set the vertical field of view for the panoramic camera.

Parameters: **- v_fov** (*float*)

set_panoramic_width()

Set the width for the panoramic camera in pixels. Height is adjusted according to vfov.

Parameters: **- width** (*int*)

set_pinhole_fov()

Set the horizontal field of view for the pinhole camera.

Parameters: **- fov** (*float*)

set_pinhole_height()

Set the height for the pinhole camera in pixels.

Parameters: **- height** (*int*)

set_pinhole_width()

Set the width for the pinhole camera in pixels.

Parameters: **- width** (*int*)

set_verbose_lvl()

Set the verbose level of the renderer. If 0: no output. If 1: Status output. If 2: statistics output. If 3: Debug output.

Parameters: **- lvl** (*int*)

9 References

[1] J. Long, A. Estey, D. Bartle, S. Olsen, and A. A. Gooch, ‘Catalyst: seeing through the eyes of a cat’, in *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, Monterey, California, 2010, pp. 116–123.

[2] J. Long and A. A. Gooch, ‘Bee prepared: Simulating bee vision in an educational game’, in *16th International Conference on Computer Games (CGAMES)*, 2011, pp. 262–269.

[3] L. Wilkins, ‘BeePilot’, *lucaswilkins.com*. [Online]. Available: http://www.lucaswilkins.com/beepilot/. [Accessed: 18-Aug-2017].

[4] N. Franceschini, J. M. Pichon, C. Blanes, and J. M. Brady, ‘From Insect Vision to Robot Vision [and Discussion]’, *Philos. Trans. Biol. Sci.*, vol. 337, no. 1281, pp. 283–294, 1992.

[5] J. Müller, ‘Familiarity in Honeybee Navigation - Behavioral and Neurocomputational Investigations’, Master Thesis, Ludwig-Maximilians-Universität München and Freie Universität Berlin, Munich, 2016.

[6] A. Borst, ‘Drosophila’s View on Insect Vision’, *Curr. Biol.*, vol. 19, no. 1, pp. R36–R47, Jan. 2009.

[7] M. V. Srinivasan and M. Lehrer, ‘Temporal acuity of honeybee vision: behavioural studies using moving stimuli’, *J. Comp. Physiol. A*, vol. 155, no. 3, pp. 297–312, May 1984.

[8] E. J. Warrant, ‘Seeing better at night: life style, eye design and the optimum strategy of spatial and temporal summation’, *Vision Res.*, vol. 39, no. 9, pp. 1611–1630, May 1999.

[9] R. Menzel, D. F. Ventura, H. Hertel, J. M. de Souza, and U. Greggers, ‘Spectral sensitivity of photoreceptors in insect compound eyes: Comparison of species and methods’, *J. Comp. Physiol. A*, vol. 158, no. 2, pp. 165–177, Mar. 1986.

[10] F. G. Varela and W. Wiitanen, ‘The Optics of the Compound Eye of the Honeybee (Apis mellifera)’, *J. Gen. Physiol.*, vol. 55, no. 3, pp. 336–358, Mar. 1970.

[11] R. Seidl and W. Kaiser, ‘Visual field size, binocular domain and the ommatidial array of the compound eyes in worker honey bees’, *J. Comp. Physiol.*, vol. 143, no. 1, pp. 17–26, Mar. 1981.

[12] W. Stürzl, N. Boeddeker, L. Dittmar, and M. Egelhaaf, ‘Mimicking honeybee eyes with a 280 degrees field of view catadioptric imaging system’, *Bioinspir. Biomim.*, vol. 5, no. 3, p. 036002, Sep. 2010.

[13] A. Giger, ‘Honeybee vision: analysis of pattern orientation’, Doctoral Thesis, Australian National University, Canberra, Australia, 1996.

[14] A. Giger, ‘BEOS’, *andygiger.com*. [Online]. Available: http://andygiger.com/science/beye/beyehome.html. [Accessed: 19-Aug-2017].

[15] German Aerospace Center, ‘Insect Vision’, *insectvision.org*. [Online]. Available: http://www.insectvision.org/. [Accessed: 19-Aug-2017].

[16] R. Menzel, K. Geiger, L. Chittka, J. Joerges, J. Kunze, and U. Müller, ‘The knowledge base of bee navigation’, *J. Exp. Biol.*, vol. 199, no. 1, pp. 141–146, Jan. 1996.

[17] S. Collins, ‘Reconstructing the Visual Field of Compound Eyes’, in *Rendering Techniques ’97*, Springer, Vienna, 1997, pp. 81–92.

[18] M. Vorobyev, A. Gumbert, J. Kunze, M. Giurfa, and R. Menzel, ‘Flowers Through Insect Eyes’, *Isr. J. Plant Sci.*, vol. 45, no. 2–3, pp. 93–101, Jan. 1997.

[19] T. R. Neumann, ‘Modeling Insect Compound Eyes: Space-Variant Spherical Vision’, in *Biologically Motivated Computer Vision*, 2002, pp. 360–367.

[20] N. L. Carreck, J. L. Osborne, E. A. Capaldi, and J. R. Riley, ‘Tracking bees with radar’, *Bee World*, vol. 80, no. 3, pp. 124–131, Jan. 1999.

[21] G. Fangi, ‘Spherical Photogrammetry for Cultural Heritage Metric Documentation: A Critical Review’, in *Built Heritage: Monitoring Conservation Management*, L. Toniolo, M. Boriani, and G. Guidi, Eds. Cham: Springer International Publishing, 2015, pp. 301–311.

[22] B. Rabus, M. Eineder, A. Roth, and R. Bamler, ‘The shuttle radar topography mission—a new class of digital elevation models acquired by spaceborne radar’, *ISPRS J. Photogramm. Remote Sens.*, vol. 57, no. 4, pp. 241–262, Feb. 2003.

[23] B. Foundation, ‘blender.org - Home of the Blender project - Free and Open 3D Creation Software’, *blender.org*. [Online]. Available: https://www.blender.org/. [Accessed: 18-Aug-2017].

[24] S. B. Laughlin and G. A. Horridge, ‘Angular sensitivity of the retinula cells of dark-adapted worker bee’, *Z. Für Vgl. Physiol.*, vol. 74, no. 3, pp. 329–335, 1971.

[25] D. Roşca, ‘New uniform grids on the sphere’, *Astron. Astrophys.*, vol. 520, p. A63, Sep. 2010.

[26] Render Legion, ‘Corona Renderer 1.6’. [Online]. Available: https://corona-renderer.com/. [Accessed: 19-Aug-2017].

[27] Intel, ‘OSPRay’. [Online]. Available: http://www.ospray.org/. [Accessed: 19-Aug-2017].

[28] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: from theory to implementation*, Third edition. Cambridge, MA: Morgan Kaufmann Publishers/Elsevier, 2017.

[29] ‘Mitsuba - physically based renderer’. [Online]. Available: https://www.mitsuba-renderer.org/. [Accessed: 19-Aug-2017].

[30] A. T. Áfra, I. Wald, C. Benthin, and S. Woop, ‘Embree ray tracing kernels: overview and new features’, 2016, pp. 1–2.

[31] Eigen Community, ‘Eigen 3.3.4.’ [Online]. Available: http://eigen.tuxfamily.org/. [Accessed: 19-Aug-2017].

[32] Intel, ‘Embree Github Tutorials’. [Online]. Available: https://github.com/embree/embree/tree/master/tutorials. [Accessed: 19-Aug-2017].

[33] J. Polster, ‘Bee Vision Simulation’, *github.com*, 17-Jul-2017. [Online]. Available: https://github.com/tschopo/bee_view. [Accessed: 19-Aug-2017].

[34] L. Tison *et al.*, ‘Honey Bees’ Behavior Is Impaired by Chronic Exposure to the Neonicotinoid Thiacloprid in the Field’, *Environ. Sci. Technol.*, vol. 50, no. 13, pp. 7218–7227, Jul. 2016.

[35] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, ‘Cython: The Best of Both Worlds’, *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 31–39, Mar. 2011.

[36] M. V. Srinivasan, ‘Pattern recognition in the honeybee: Recent progress’, *J. Insect Physiol.*, vol. 40, no. 3, pp. 183–194, Mar. 1994.

[37] A. M. Anderson, ‘Shape perception in the honey bee’, *Anim. Behav.*, vol. 25, pp. 67–79, Feb. 1977.

[38] C. Benthin, S. Woop, M. Nießner, K. Selgrad, and I. Wald, ‘Efficient Ray Tracing of Subdivision Surfaces using Tessellation Caching’, in *Proceedings of the 7th High-Performance Graphics Conference*, 2015.