S3 Text: Geometry and software control of a spherical visual stimulation arena with square LED tiles

originally published as supplementary material to Dehmelt, Meier, Hinz et al., bioRxiv 2019, last edited on September 1, 2019

The purpose of this document

During methods development, two major concerns surfaced early on: Drafting the structural parts in CAD software prior to printing required an exact (or at least approximate) definition of their shape and location. And displaying visual stimuli in the arena made it necessary to translate the desired image into accurately timed instructions to each one of the 7,552 individual LEDs, depending on their actual location in three-dimensional space. We here present a precise solution to both problems. This document was originally intended to provide new lab members a self-contained introduction to these ideas, and is thus written in the style of a manual for undergraduates. It contains a mathematical description of the geometry of the arena architecture, as was used for printing. It further contains a mathematical description of the position in space of each individual LED, and the mapping between stimulus space and physical space. Finally, we appended an example of MATLAB code performing this mapping. A mathematical appendix reviews some of the concepts underlying coordinate transformations.

Table of contents

1 Geometry of the structural backbone

1.1 Cartesian and geographic coordinate systems

A position in three-dimensional space can be described by an infinite variety of coordinate systems, affording us the privilege of picking whichever we find the most convenient. Throughout this manuscript, we will be using either Cartesian or geographic coordinates. The former streamlines the description of simple translations as well as the shape of ”flat” surfaces, whereas the latter efficiently captures the geometry of a sphere surface with just two of its three coordinates[1]. Fig. 1 illustrates both systems, and introduces our convention for the respective coordinates: x, y and z in the Cartesian system, as well as azimuth α, elevation β and radius r in the geographic system. It is worth noting that, as opposed to the most common form of polar coordinates, geographic coordinates assign an elevation of +90° to the ”north pole” of a sphere, and -90° to its ”south pole”, rather than an inclination from 0° to 180°. The azimuth is the same as for default polar coordinates, ranging from -180° to +180°.

When we keep using more than one coordinate system at a time, there is always some risk of confusion. But there are ways to point out that when we wrote down a certain set of coordinates, we actually had a specific coordinate system in mind. See appendix A.1 for details.

[1]In fact, we use a specific type of geographic coordinates, known as Up-West-North or UWN geographic coordinates. As zebrafish will virtually always be placed upright into the arena, this system is consistent with a common way of describing rightward zebrafish eye movement as ”positive”, and leftward movement as ”negative”. Because many experimenters are used to observing their animals from above, this is otherwise known as the ”clockwise” notation. Please note that the inverse convention, known as ”counter-clockwise” or even ”mathematically positive”, is also widespread.

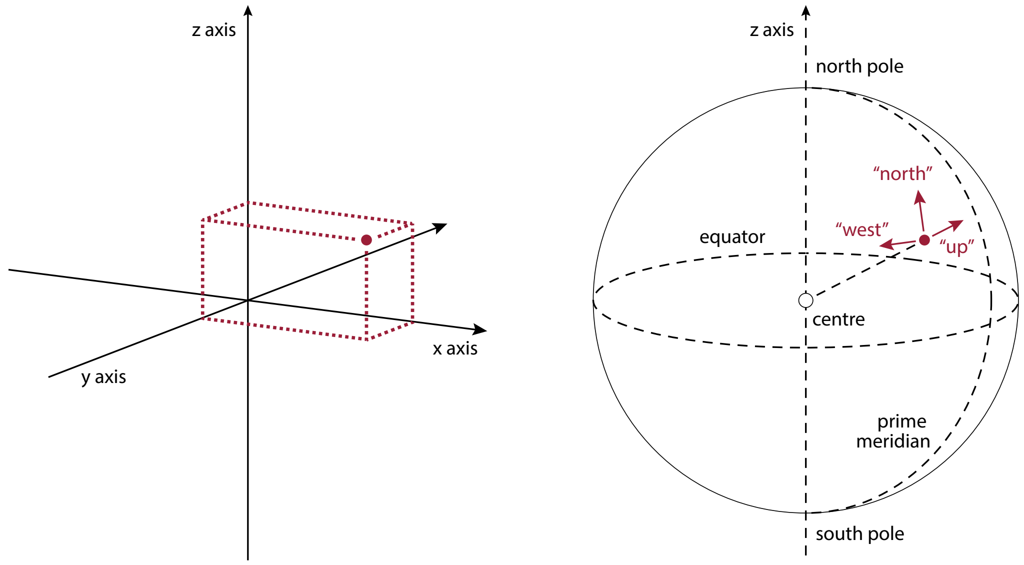


Figure 1: Choice of coordinate systems. (a) In red, the position of a point can be given in Cartesian coordinates with $x, y, z \in \,]-\infty, \infty[$. (b) The same point can also be described as sitting on the surface of a sphere centred on the origin. In one possible incarnation of geograpic coordinates, so-called UWN geographic coordinates, its position is then given by how far "up" it is from the sphere centre, how far "west" it is from the prime meridian, and how far "north" from the equator. These three coordinates are known as radius r, azimuth α and elevation β.

No matter which coordinate systems we use, and no matter how we present their coordinates in practice, each point in physical space always has exactly one correct and unambigious[2] description in each system. And we can always unambiguously convert the description of a point p, written down with respect to one coordinate system, into its description with respect to the other coordinate system. This is rather intuitive: Just because we switch our way of describing positions, the positions themselves do not change. For instance, the following equations convert geographic coordinates into Cartesian coordinates without any loss of information:

$$\begin{aligned} x &= r \cos\alpha \cos\beta \\ y &= r \cos\alpha \sin\beta \\ z &= r \sin\alpha \end{aligned} \tag{1}$$

We can also go in the opposite direction, taking a description in Cartesian coordinates and converting it to geographic coordinates:

[2]There are some exceptions to this rule. In polar coordinate systems, the point of origin can be described by a radius of zero, and any combination of angles. In geographic coordinates, the poles of a sphere are described by some fixed radius, an elevation of +90° or −90° respectively, and any azimuth whatsoever. Furthermore, there is no difference between an azimuth of exactly +180° and one of exactly −180° so we generally limit the azimuth to $\alpha \in \,]-180, 180] \subset \mathbb{R}$. All of these description are still "unambiguous", in that each one of them points to exactly one point. But they are no longer "unique", because we can choose between different ways of a addressing the same point. Fortunately, none of these exceptions are likely to be relevant in practice.

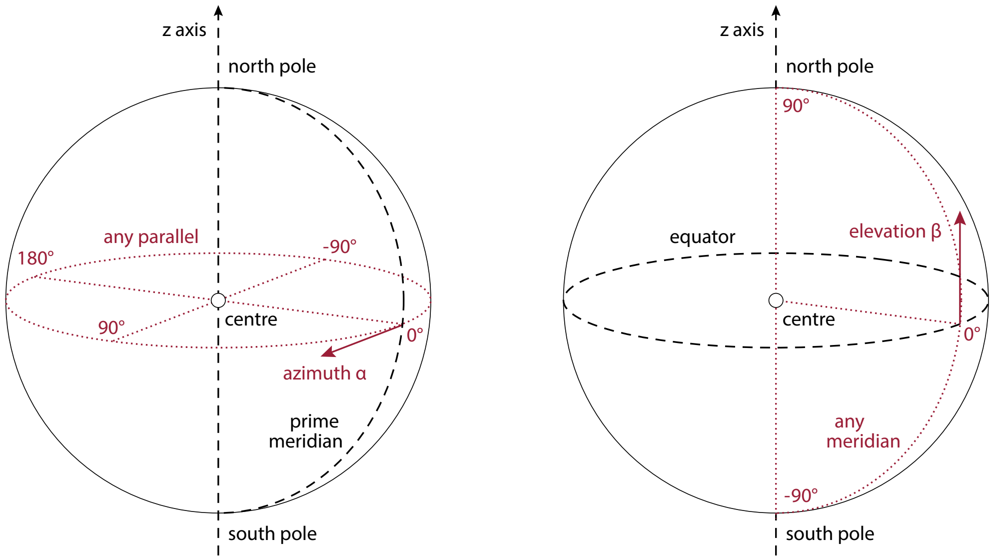


Figure 2: Choice of coordinate systems. (a) The range of azimuth values is $\alpha \in [-180, 180[$. (b) The range of elevation angles is $\beta \in [-90, 90[$ and $r \in [0, \infty[$.

$$
\begin{aligned}
r &= \sqrt{x^2 + y^2 + z^2} \\
\alpha &= \sin^{-1} \frac{z}{r} = \sin^{-1} \left(\frac{z}{\sqrt{x^2 + y^2 + z^2}} \right) \\
\beta &= \cos^{-1} \left(\frac{x}{r \cos \alpha} \right) = \cos^{-1} \left(\frac{x / \sqrt{x^2 + y^2 + z^2}}{\cos \left(\sin^{-1} \left(z / \sqrt{x^2 + y^2 + z^2} \right) \right)} \right)
\end{aligned}
\tag{2}
$$

For further details, please refer to appendix A.2. Individual LEDs come arranged on square tiles (Fig. 6), which in turn are distributed across the spherical arena. To determine the position of each individual LED, we will need to compare the shape and extent of both ”round” and ”flat” objects in space. This is where the unit vectors of both coordinate systems come in. The relevant geographic unit vectors, expressed in Cartesian coordinates, are

$$
\mathrm{u}_\alpha = \frac{1}{r} \cdot \frac{\partial}{\partial \alpha} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\cos \alpha \sin \beta \\ +\cos \alpha \cos \beta \\ 0 \end{pmatrix} \tag{3}
$$

which is always tangential to a line of longitude, pointing ”westward”, and

$$
\mathrm{u}_\beta = \frac{1}{r} \cdot \frac{\partial}{\partial \beta} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\sin \alpha \cos \beta \\ -\sin \alpha \sin \beta \\ \cos \alpha \end{pmatrix} \tag{4}
$$

which is always tangential to a circle of latitude, pointing ”northward”. For an intuition on unit vectors, as well as a complete mathematical derivation, see appendix A.3. Lines of longitude are also called ”meridians”, with the ”prime meridian” at $\alpha = 0$ (Fig. 1f). Circles of latitude are sometimes called ”parallels”; the parallel at $\beta = 0$ is known as the ”equator” (Fig. 1e).

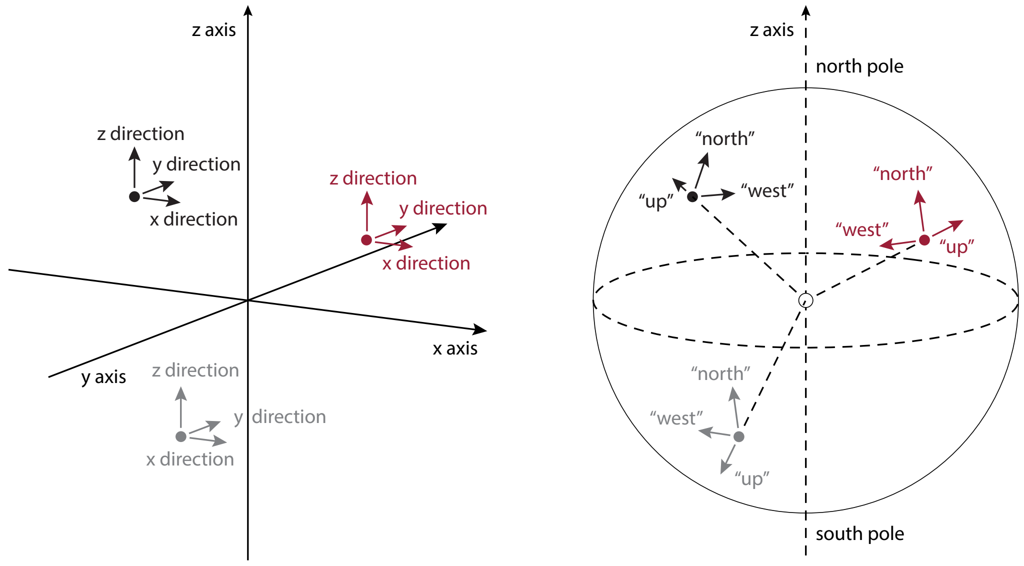


Figure 3: Unit vectors describe the incremental change of coordinates. (a) The direction of unit vectors such as $u_x = \partial p/\partial x$ is always constant in Cartesian space, regardless of position p, whereas (b) the direction of unit vectors $u_\alpha = u_\alpha(\alpha, \beta) = \partial p/\partial\alpha$ and $u_\beta = u_\beta(\alpha, \beta) = \partial p/\partial\beta$ depends on the actual value of both angular coordinates. It is however independent of radius r. For a detailed explanation, see appendix A.3.

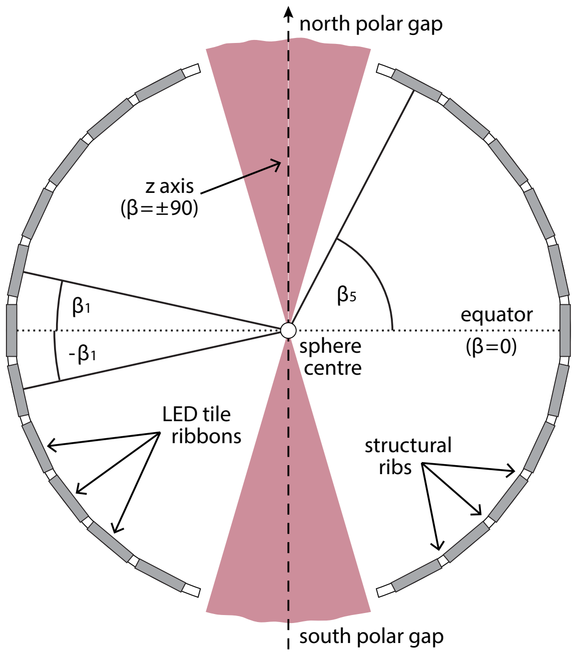
1.2 Physical and hardware constraints on arena size

Our hardware controller and C code can drive up to 240 LED tiles with 64 LEDs each. With these constraints, we need to identify the optimal size of the arena to build. In the end, the inner edges of all structural ribs will lie on a sphere with radius R_S, a radius we can choose. LED tiles will be arranged in latitudinal ribbons, and the number of such ribbons depends on the radius of the sphere. In this section, we will

- Identify a plausible range of radii.
- Choose a compatible number of ribbons.
- Use the number of ribbons to narrow down the radius.
- Add a safety margin to this radius to accomodate unavoidable gaps.

Readers who do not seek details on these estimates can safely skip this section. With dimensions of 20 mm by 20 mm, each tile has an area of 400 mm^2, adding up to 96 000 mm^2 for all tiles combined. To optimally cover a sphere with these tiles, said sphere should have a somewhat larger area to accomodate to square shape of the tiles (which will lead to unavoidable gaps in between them), as well as the need for structural elements carrying the weight of the assembly. Sphere radius R_S and area A_S are linked via $A_S = 4\pi R_S^2$. This imposes a strict lower limit on the sphere radius, $R_S > 87.4$ mm. With a generous margin to account for large and small gaps, imprecise manufacturing and manual placement, we obtain a more plausible range of $R_S \in [90\,\text{mm}, 110\,\text{mm}]$. To further narrow down the radius, we need to select the exact number of ribbons to construct.

The number of ribbons we can fit onto the surface depends on the size of each ribbon and additional elements such as structural ribs and gaps (Fig. 4). These calculations are simpler if the radius of the sphere is much larger than the width of the tiles. This is true for very large spheres, or very narrow tiles. Neither is the case for our arena, but it is a good first estimate. Later on, we will add an additional margin to the radius. Let us consider the height of our tiles, as they are placed



ribbon no.	tiles per ribbon	elevation of tile centre
5	2×5	$\beta_5 = 62.4$
4	2×9	$\beta_4 = 49.9$
3	2×11	$\beta_3 = 37.6$
2	2×13	$\beta_2 = 25.0$
1	2×14	$\beta_1 = 12.5$
0	2×15	$\beta_0 = 0$
−1	2×14	$-\beta_1 = -12.5$
−2	2×13	$-\beta_2 = -25.0$
−3	2×11	$-\beta_3 = -37.6$
−4	2×8	$-\beta_4 = -49.9$
−5	2×5	$-\beta_5 = -62.4$

Figure 4: Arrangement of tiles into ribbons of equal elevation. Cross-section of the arena. Values of β have been rounded to one decimal place.

right above one another along one of the meridians of the sphere. Circumnavigating this meridian from pole to pole and back again, we would cover an angle of 360°. On this circular trajectory, we would twice encounter each ribbon of LED tiles and each structural rib, and we would also encounter any polar holes in the sphere. In our case, there is one large hole of about 30° around each pole to allow for an optical path to be coupled in. These holes correspond to elevations of $\beta \in [-90, -75]$ and $\beta \in [75, 90]$, respectively. Because we plan to arrange all LED tiles symmetrically, it is sufficient to consider only the trajectory from one pole to the other, corresponding to a 180° range. If there are no additional gaps, the individual elements then add up as follows:

$$\begin{aligned} 180° &= \hat{\beta}_{\text{all tiles}} + \hat{\beta}_{\text{all ribs}} + \hat{\beta}_{\text{hole}} \\ &= N_{\text{ribbon}} \cdot \hat{\beta}_{\text{tile}} + (N_{\text{ribbon}}+1) \cdot \hat{\beta}_{\text{rib}} + \hat{\beta}_{\text{hole}} \\ &= N_{\text{ribbon}} \cdot 2\sin^{-1}\left(\frac{d_{\text{tile}}}{2R_S}\right) + (N_{\text{ribbon}}+1) \cdot 2\sin^{-1}\left(\frac{d_{\text{rib}}}{2R_S}\right) + 2\sin^{-1}\left(\frac{d_{\text{hole}}}{2R_S}\right) \end{aligned} \tag{5}$$

where $\hat{\beta}$ designates the angular size of each element in terms of elevation. We picked a conservative estimate of the edge of each tile around $d_{\text{tile}} = 21$ mm instead of 20 mm to account for imprecise placement, and chose the thickness of structural ribs to be one tenth of that, i.e., $d_{\text{rib}} = 2.1$ mm. Substituting the minimum and maximum from the range of plausible radii above, we can solve this equation twice to find that we could fit either 10, 11 or 12 ribbons. To have an uninterrupted ribbon of LEDs right around the equator of the sphere, we opted to include an odd number of ribbons ($N_{\text{ribbon}} = 11$), with 5 ribbons above and 5 ribbons below the equator. Based on this number, we computed a more precise estimate of the optimal sphere radius by numerically solving equation (5), obtaining $R_S = 101$ mm.

This radius would be accurate if the horizontal extent of the tiles were much smaller than the radius of the sphere. However, they are in fact square, and their width is only one order of magnitude smaller than the approximate diameter of the sphere. This means that tiles facing the sphere centre

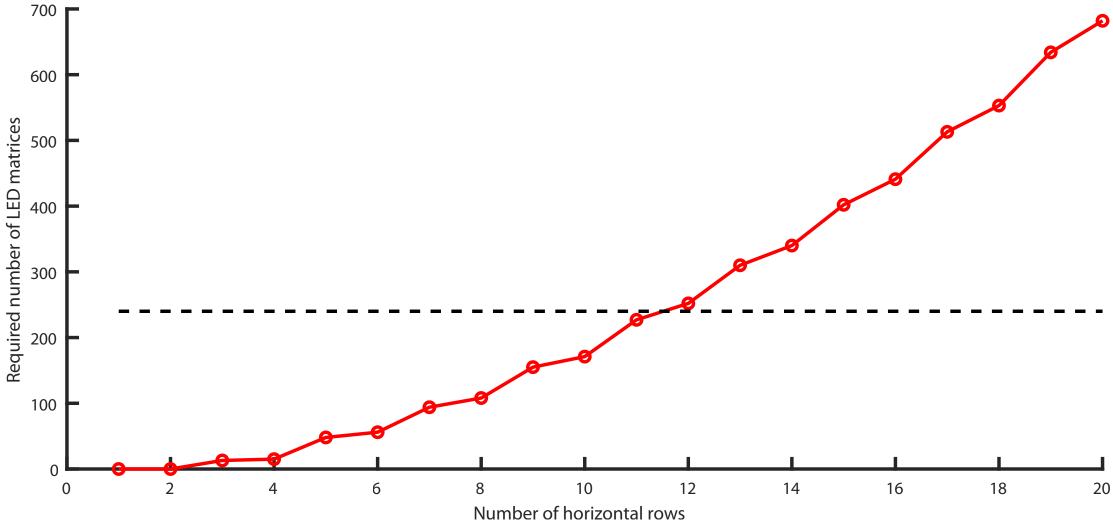


Figure 5: Relationship between the number of LED ribbons in a given arena design and the total number of LEDs required for maximum coverage. Figure provided by Julian Hinz.

such that their edges are parallel to the α and β directions have a larger angular size the closer they are to one of the poles. Thus, estimating their angular size as $2\sin^{-1}(d_{\mathrm{rib}}/2R_S)$, as we did in equation (5), is insufficient. To spread out the structural ribs further, Julian Hinz chose to print all parts scaled by an additional factor of 1.05, effectively increasing the sphere radius to 106.5 mm. This way, the difference in elevation between neighbouring ribbons is the same, no matter whether we compare the equatorial ribbon to the one below, or the two top-most ribbons to one another. This extra space is sorely needed for the ribbons near the poles, and is still small enough not to be too wasteful for nearly equatorial ones. The ”inner” sphere radius computed here corresponds to the distance between fish and stimulus, although some LEDs (notably those near the centre of the tiles), are located marginally closer. Where structural and electronic elements protrude from the arena, the outer perimeter of the sphere is considerably larger, reaching points as far as 150 mm.

1.3 Alternative designs

Readers interested in designing larger or smaller spherical arenas may consider how the total number of LED tiles required to achieve maximum coverage is linked to the number of LED ribbons (Fig. 5).

2 Geometry of the LED arrangement

2.1 Placement of LED tiles

The arrangement of LED tiles into ribbons is shown in Fig. 4, which also lists their respective elevations. Based on the suspicion that frontal stimuli are more behaviourally relevant to zebrafish than those in the rear, we chose to align all LED ribbons with the frontal keel to minimise the gaps there. The first tile of each ribbon is located immediately next to this keel (α_C near zero), and each following tile is placed next to the existing ones, with increasingly large absolute values of azimuth. Both hemispheres of the arena are populated symmetrically from the keel.

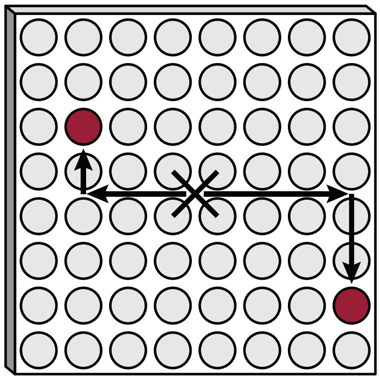


Figure 6: Individual LED position relative to tile centre. Each tile carries 64 individual LEDs in an eight-by-eight pattern. The position of each individual LED is equal to that of the tile centre, plus a weighted sum of unit vectors (black) spanning the tile surface. Because each tile faces the centre of the sphere, the increment unit vectors $\mathrm{u}_\alpha(\alpha_C, \beta_C)$ and $\mathrm{u}_\beta(\alpha_C, \beta_C)$ defined at the tile centre, $\mathrm{p}_C = (r, \alpha_C, \beta_C)$, span the tile surface (cf. appendix A.3). Distance between the centres of nearest neighbours is $d_{\mathrm{LED}} = 2.48\,\mathrm{mm}$.

2.2 Resulting position of individual LEDs

Each tile carries 64 individual LEDs arranged in a square eight-by-eight pattern (Fig. 6). From section 2.1, we already know the position of the centre of each tile. Now we must figure out where each individual LED is located. Because the tiles are flat and face the sphere centre, all LEDs lie further away from sphere centre than their tile centre. Their geographic coordinates are non-trivial, as – seen from the sphere centre – LEDs near the edge of a tile appear closer to their neighbours than those near the centre of the tile. However, we can compute the exact geographic coordinates of each individual LEDs by combining the location of the centre of the tile holding them with what we know about the distribution of LEDs across the flat tile. Because each tile faces the centre of the sphere, the increment unit vectors u_α and u_β, evaluated at the tile centre, span its surface. Thus, the position of each individual LED held by a tile is equal to that of the tile centre, plus a weighted sum of these unit vectors (Fig. 6). As illustrated in Fig. 7, we must

1. Convert tile centres from geographic coordinates $\mathrm{p}_C = (R_S, \alpha_C, \beta_C)$ to Cartesian $\mathrm{p}_C = (x_C, y_C, z_C)$.
2. Add scaled versions of the increment unit vectors u_α and u_β, also in Cartesian coordinates.
3. Convert the result back to geographic coordinates.

Performing the first and second step, we find that the absolute position p_{ij} of each individual LED in Cartesian three-dimensional space is given by

$$\mathrm{p}_{ij} = \mathrm{p}_C \;+\; (i-4.5)\, d_{\mathrm{LED}} \cdot \mathrm{u}_\alpha(\alpha_C, \beta_C) \;+\; (j-4.5)\, d_{\mathrm{LED}} \cdot \mathrm{u}_\beta(\alpha_C, \beta_C) \tag{6}$$

where $i, j \in [1, 8] \subset \mathbb{N}$, and $d_{\mathrm{LED}} = 2.48\,\mathrm{mm}$ is the distance between the centres of nearest neighbours. With equations (1), (3) and (4), this becomes

$$\mathrm{p}_{ij} = R_S \begin{pmatrix} \cos\alpha_C \cos\beta_C \\ \cos\alpha_C \sin\beta_C \\ \sin\alpha_C \end{pmatrix} + (i\text{–}4.5)\, d_{\mathrm{LED}} \begin{pmatrix} -\cos\alpha_C \sin\beta_C \\ \cos\alpha_C \cos\beta_C \\ 0 \end{pmatrix} + (j\text{–}4.5)\, d_{\mathrm{LED}} \begin{pmatrix} -\sin\alpha_C \cos\beta_C \\ -\sin\alpha \sin\beta_C \\ \cos\alpha_C \end{pmatrix} \tag{7}$$

In a third step, we convert these Cartesian coordinates back into geographic coordinates using equation (2). Fig. 8 reveals the idealised coordinates of all individual LEDs. In the actual process of construction, deviations on the order of $\pm 1°$, especially in α_C, are hard to avoid.

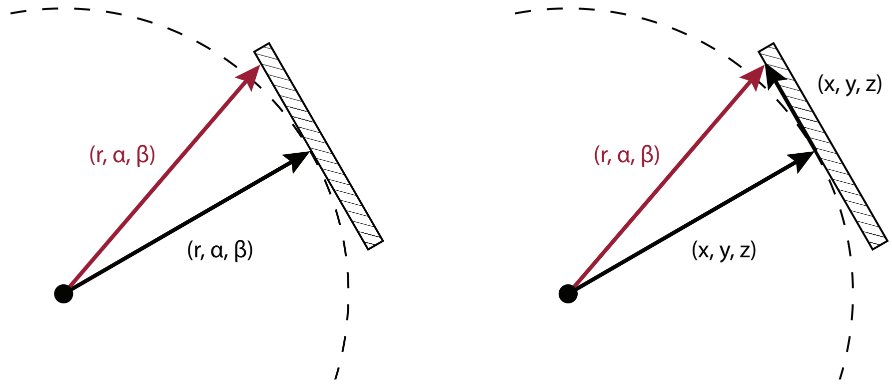


Figure 7: Individual LED coordinates. (Left) We can compute the unknown geographic coordinates of individual LEDs, in red, from the location of the centre of the tile holding them, in black. (Right) To do so, we must convert tile centre position to Cartesian coordinates, add scaled versions of the increment unit vectors $u_\alpha(\alpha_C, \beta_C)$ and $u_\beta(\alpha_C, \beta_C)$, also in Cartesian coordinates, then convert the result back to geographic coordinates.

2.3 A note on limitations and artifacts

The setup as described above is versatile stimulation arena made from a combination of 3D-printed parts and relative low-cost products such as the square LED tiles. By its very architecture, it does however have a number of undesirable properties.

Gaps. Due to the nature of the mathematical challenge of fitting square tiles onto a spherical surface, gaps in between tiles are unavoidable. Others result from structural elements such as the ”keel” along the prime meridian that does not carry LEDs themselves. The reduced number of LEDs per unit of surface area leads to lower total luminance of a stimulus presented there, and the animal might perceive the dark triangles themselves as objects, distracting its attention from the stimulus itself. Over the course of her thesis project, Rebecca Meier performed several control experiments, none of which revealed such detrimental effects. She attached hand-crafted shapes across more densely covered parts of the arena, obscuring existing LEDs with dark triangles or dark bars assimilating the gaps in between LEDs as well as structural elements. Other control experiments are in preparation.

Resolution and smoothness. The edges of vertical bars are obviously vertical, but the grid of LEDs on each tile is not, unless said tile is located on the equator. This leads to a distortion of the shape of the bar near the poles, where instead of a straight line of LEDs above one another, it will be represented by a stair-shaped succession of LEDs. This is further aggravated by the relative low number of LEDs in total, leading to noticeable pixelation of the stimulus. However, larval zebrafish have poor visual acuity, with an angular resolution of only about 2° (Aristides B. Arrenberg, personal communication). This makes such limitations unlikely to have an effect for all but the most extreme stimulus positions. Control experiments pending.

Countermeasures. To alleviate some of these concerns, users may want to consider placing a layer of diffusors onto the LED layer. In the Arrenberg lab, we have successfully used such diffusors to stimulate fish under a two-photon microscope, using an arena with significantly fewer LEDs. The geometry of the sphere, however, makes designing and installing such diffusors non-trivial.

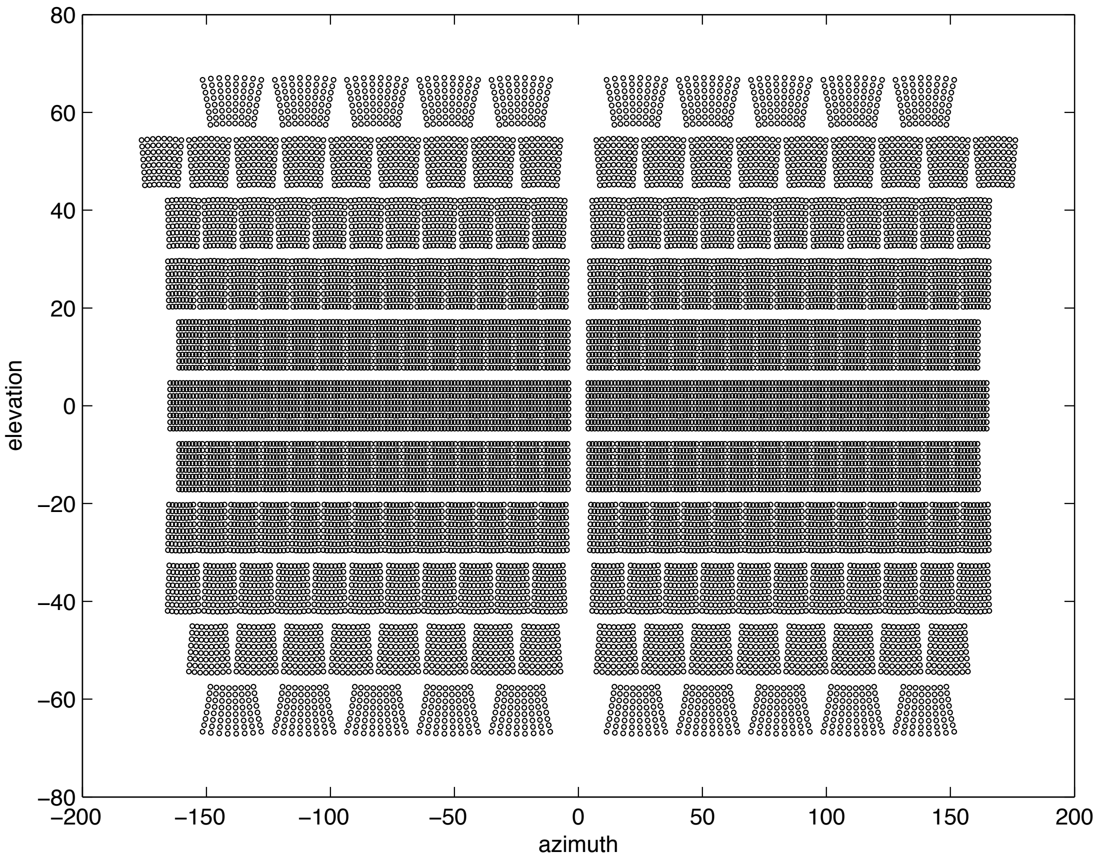


Figure 8: Individual LED positions in geographic coordinates. Each circle represents a single LED. Each cohesive group of eight-by-eight circles corresponds to the 64 LEDs contained in a single tile. Most of the sphere surface is densely covered by LEDs, but there are several regions with important gaps. These are generally located in parts of the visual field suspected to be less behaviourally relevant, but others were simply unavoidable due to mechanical constraints on the setup. Even where an LED-holding tile is placed, the angular gap between LEDs increases the closer the tile is to either pole of the sphere. Likewise, nearly triangular gaps in between tiles become more apparent towards the poles. The left and right hemispheres are symmetrical but for unavoidable mechanical imprecisions during assembly. Top and bottom hemispheres are almost symmetrical, except for additional tiles added to the +50° ribbon near the top, one on the left and one on the right. These are absent from the corresponding −50° ribbon near the bottom, due to physical constraints on our specific setup.

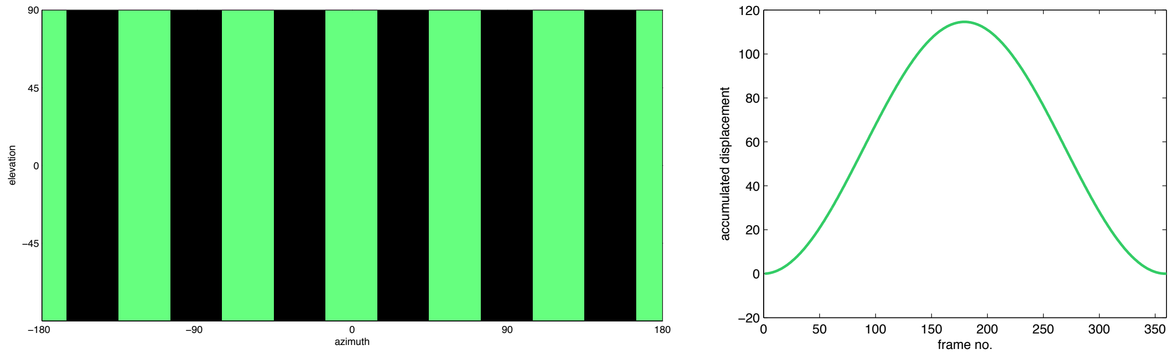


Figure 9: Mapping visual stimuli onto LED space. (a) Visual stimulus in geographic coordinates. (b) Common sinusoidal velocity profile.

3 Mapping visual stimulation onto physical space

3.1 Moving-bar stimuli in geographic coordinates

Visual stimulation in the Arrenberg lab is often presented in the form of horizontally moving gratings, i.e., horizontally moving vertical bars of equal width, and distance from one another. When shown on a flat screen, these are indeed bars of constant width, as measured in millimetres. On a spherical surface, a pattern resembling a ”beach ball” is used instead, where the width of each bar still covers a constant angular range; its width as measured in millimetres, however, decreases towards the poles of the sphere. The shape of such a stimulus, as seen from the centre of the sphere, is shown in Fig. 9. Note that this stimulus is shown in geographic coordinates, so its ”beach ball” shape is not immediately apparent. If the poles themselves were covered by LEDs, all bars would meet there in a single point. In practice, the poles are often left open to couple in an optical path.

3.2 Cropping the shape of moving-bar stimuli

Stimuli can be cropped to be displayed on one of the eight hemispheres only, or on icosahedrally distributed circular areas equidistantly covering the sphere surface. Code for this cropping is available separately.

3.3 Mapping individual frame content to LED positions

Knowing what the stimulus should look like to the animal at the centre of the sphere, we still need to make sure this stimulus is indeed generated by our assembly of LEDs. In other words, we must translate the desired shape of the stimulus, provided in spatial coordinates using whatever coordinate system we find convenient, into a set of on-or-off instructions to each individual LEDs at every point in time. This is akin to mapping the spatial description of the stimulus onto the spatial coordinates of each LED. To the hardware controller, however, each LED is known by a systematic address that has little or no relation to its physical location. So we must additionally create a look-up table that matches each LED address with the corresponding spatial coordinates of that same LED. Neither of these steps is very complicated from a mathematical point of view, but it has to be taken with care to avoid a garbled stimulus.

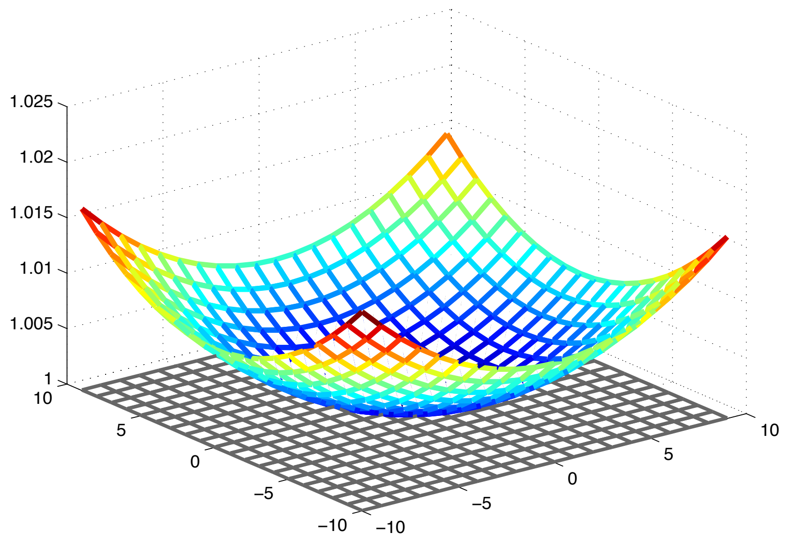


Figure 10: Fraction of sphere surface covered by a square LED tile. The segment of a sphere surface (shown in colour) above any perpendicular flat square (shown in grey) can be computed analytically using equation 8. To compute the effective surface area covered by an LED tile, as seen from the sphere centre, we must not use the the LED tile itself as the perpendicular square shown here in grey. The square immediately below the surface segment, while located in the same place as the LED tile, is in fact slightly smaller (cf. Fig. ??).

3.4 Surface coverage

The area of a surface segment delimited by the projection of the edges of a single tile onto the sphere centre is given by

$$A(S) = \oint dA = \int_{-\lambda}^{+\lambda} dx \int_{-\lambda}^{+\lambda} dy \, \|\mathrm{u}_\alpha \times \mathrm{u}_\beta\| = \int_{-\lambda}^{+\lambda} dx \int_{-\lambda}^{+\lambda} dy \, \frac{R_S^2}{R_S^2 - x^2 - y^2} \tag{8}$$

where $(\pm\lambda, \pm\lambda)$ is the Cartesion position of the edges of a smaller rectangle, which is the straight projection of the sphere segment onto the tile,

$$\lambda = \sin\left(\tan^{-1}\left(\frac{d}{2R_S}\right)\right) \tag{9}$$

These equations can be used to estimate the total coverage of the sphere by its square LED tiles. Pooling over the entire sphere, including all gaps, holes and structural elements, we obtain a coverage of 66.54%. Locally, coverage is much higher, with the equatorial ribbon reaching 80% if the large rear hole and structural elements are included, and well above 90% if only the key parts of the visual field are taken into account.

A Mathematical appendix

A.1 Vector notation, and user confusion

When we keep using more than one coordinate system at a time, there is always some risk of confusion. But there are ways to point out that, when we wrote down a certain set of coordinates, we actually had a specific coordinate system in mind:

- Using variables. We can list the coordinates individually, referring to them by their mathematical variable name, such as "the point identified by $r = 5$, $\alpha = 30$ and $\beta = -20$". This is not a particularly compact, but fairly safe solution. We just have to make sure that different coordinate systems use different variable names.

- Using words. We can avoid confusion by verbally identifying the coordinate system, as in

  $$\text{"...some point p} = \begin{pmatrix} 5 \\ 30 \\ -20 \end{pmatrix} \text{ in geographic coordinates..."}$$

  In this case, it is particularly important to use an unambiguous name for our coordinate system; while geographic coordinates are a type of "polar coordinates", they are not the standard type of polar coordinates. Because column vectors take up a lot of space on paper, we often write them as transposed row vectors instead, such as $\mathrm{p} = (5, 30, -20)^{\mathrm{T}}$.

- Using labels. If we need our notation to be compact, we can add labels indicating the coordinate system behind the numbers. For instance, writing $[\mathrm{p}]_G = (5, 30, -20)^{\mathrm{T}}$ instead of just p, we know that these are geographic coordinates, where $r = 5$, $\alpha = 30$ and $\beta = -20$. This is a little tedious, but it can be safer than just writing $\mathrm{p} = (5, 30, -20)^{\mathrm{T}}$, which might be interpreted as $x = 5$, $y = 30$ and $z = -20$, a very different point in space.

- Using context. For maximum efficiency, we can rely on context alone to provide enough information for readers to tell the coordinate systems apart. This is what we do in our labs most of the time, and it turns out to be less confusing than it might seem.

A.2 One point, many addresses. Converting between coordinate systems

No matter which coordinate systems we use, and no matter how we present their coordinates in practice, each point in physical space always has exactly one correct and unambigious[3] description in each system. And we can always unambiguously convert the description of a point p, written down with respect to one coordinate system, into its description with respect to the other coordinate system. This is rather intuitive: Just because we switch our way of describing positions, the positions themselves obviously do not change. For instance, the following equations convert geographic coordinates $[\mathrm{p}]_G = (r, \alpha, \beta)^{\mathrm{T}}$ into Cartesian coordinates $[\mathrm{p}]_C = (x, y, z)^{\mathrm{T}}$ without any loss of information:

$$\begin{aligned} x &= r \cos\alpha \cos\beta \\ y &= r \cos\alpha \sin\beta \\ z &= r \sin\alpha \end{aligned} \tag{10}$$

[3]There are some exceptions to this rule. In polar coordinate systems, the point of origin can be described by a radius of zero, and any combination of angles. In geographic coordinates, the poles of a sphere are described by some fixed radius, an elevation of $+90°$ or $-90°$ respectively, and any azimuth whatsoever. Furthermore, there is no difference between an azimuth of exactly $+180°$ and one of exactly $-180°$ so we generally limit the azimuth to $\alpha \in]-180, 180] \subset \mathbb{R}$. All of these description are still "unambiguous", in that each one of them points to exactly one point. But they are no longer "unique", because we can choose between different ways of a addressing the same point. Fortunately, none of these exceptions are likely to be relevant in practice.

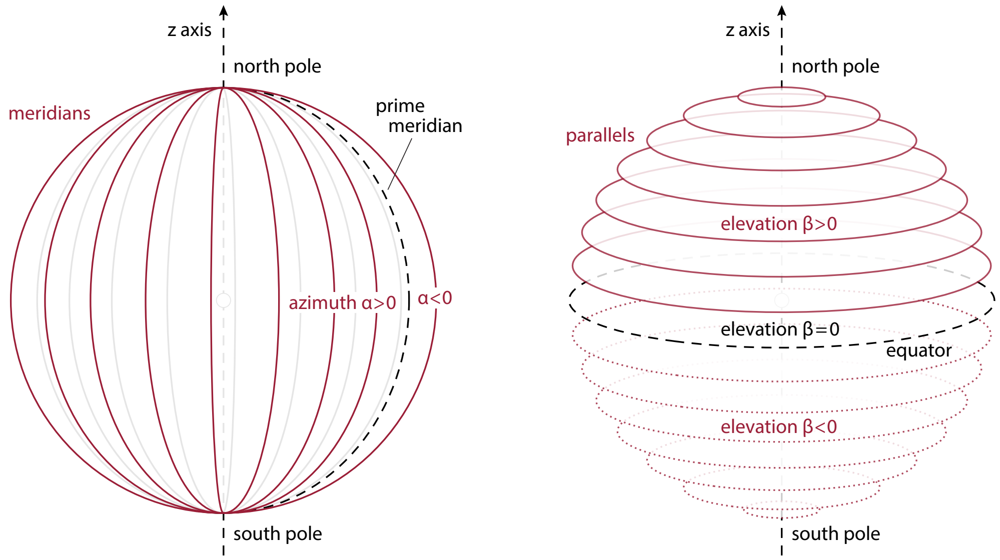


Figure 11: UWN geographic coordinates. (a) All points sharing a specific radius r and azimuth α lie on the same meridian, a hemicircle from pole to pole. Dashed lines indicate negative values of alpha. (b) All points sharing a specific radius r and elevation β lie on the same parallel, a circle parallel to the equator.

In most cases, both of our coordinate systems can be inferred from context and from the variables present in the equation. We will thus omit notations like $[\mathrm{p}]_C$ or $[\mathrm{p}]_G$ from now on, and write p instead, whatever the coordinate system used. We can also go in the opposite direction, taking a description in Cartesian coordinates and converting it to geographic coordinates:

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \alpha &= \sin^{-1} \frac{z}{r} = \sin^{-1} \left(\frac{z}{\sqrt{x^2 + y^2 + z^2}} \right) \\ \beta &= \cos^{-1} \left(\frac{x}{r \cos \alpha} \right) = \cos^{-1} \left(\frac{x / \sqrt{x^2 + y^2 + z^2}}{\cos \left(\sin^{-1} \left(z / \sqrt{x^2 + y^2 + z^2} \right) \right)} \right) \end{aligned} \tag{11}$$

This is not a very pretty set of equations, but it gets the job done: If we know the x, y and z coordinates of a point, we can now compute the equivalent r, α and β coordinates.

A.3 Unit vectors

An intuition for unit vectors

On top of just being alternative addressing systems describing the same points using different numbers, Cartesian and geographic coordinates differ in more subtle ways. One of these differences – the so-called unit vectors – become important in section 2.2, where we derive the placement of individual LEDs relative to the centre of the tile holding them. For now, let's revisit the basics.

In Cartesian coordinates, incremental changes of just one variable always move a point by a fixed distance in a fixed direction. Imagine your setup is contained in a big cardboard box. Maybe your y axis points from the left end of your setup to the right end of your setup. Then, increasing the y coordinate of a point by a tiny bit, (e.g., from 7 to 7.1) will always move your point a tiny bit towards the right-hand wall of the cardboard box (e.g., by 1mm). It does not matter whether this point was originally near the top or the bottom of the box, already close to the right-hand wall of the box, or very far away from it. In other words, it does not matter what its x or z coordinate is. This seems so natural we would not usually waste any time thinking about it. Increasing y a tiny bit always means "move this point a tiny bit towards the right-hand wall".

But in geographic coordinates, incremental changes of just one variable have a very different effect: Increasing α means moving your position along some circle in space, and it's an entirely different circle depending on the point you're moving. Increasing α might move a point over large or small distances, closer to the right wall or closer to the left, as well as closer to the rear wall or closer to the front. It all depends on where exactly that point was originally located. This sounds confusing. But if we happen to have an object in our setup that is shaped like a circle or a sphere (some people might even want to build a spherical stimulation arena), using a geographic coordinate system instead of a Cartesian one actually makes our lives easier, not harder.

Let us consider two examples. We can describe any position p in three-dimensional space using Cartesian coordinates, $\mathrm{p}=(x, y, z)^{\mathrm{T}}$, as shown in Fig. 3a. Let us look at three points in this space.

For each point in space, we can predict the direction in which a point would move if any one coordinate were to increased incrementally. To illustrate this effect, we could draw a vector between the old and new position of the point. But because the changes are tiny, this vector would be nigh-invisible. Instead, we will draw a vector that, while pointing in the direction of the incrediby tiny change, is in fact much longer than that. The exact length doesn't matter, because we only really care about the direction. To keep things simple and easy to memorise, we will usually choose a length of exactly 1, which is why these vectors are called "unit vectors".

Now, we can look at the same three points again, this time describing their position with geographic coordinates $\mathrm{p} = (r, \alpha, \beta)^{\mathrm{T}}$. We can once again determine the direction an incremental change of one coordinate would take us from each of these points. But this time, something has changed: the unit vectors point in different direction, depending on which point we compute them at. This is a fundamental property of geographic coordinates, and polar coordinates in general. It may seem confusing at first, but if used correctly, it allows us to solve problems that would otherwise be hard to figure out. One such example is fitting squares onto a sphere, and determining the geographic coordinates of different points within each square – or in our case, determining the geographic coordinates of individual LEDs (section 2.2).

Computing unit vectors

The Cartesian unit vectors, expressed in Cartesian coordinates, are

$$\begin{aligned}
\mathrm{u}_x &= \frac{\partial}{\partial x}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\
\mathrm{u}_y &= \frac{\partial}{\partial y}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \\
\mathrm{u}_z &= \frac{\partial}{\partial z}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}
\end{aligned} \tag{12}$$

These vectors each have length one, befitting a unit vector. The geographic unit vectors, expressed in Cartesian coordinates, can be computed from

$$\begin{aligned}
\frac{\partial}{\partial r}\begin{pmatrix} x \\ y \\ z \end{pmatrix} &= \begin{pmatrix} r\cos\alpha\cos\beta \\ r\cos\alpha\sin\beta \\ r\sin\alpha \end{pmatrix} \\
\frac{\partial}{\partial \alpha}\begin{pmatrix} x \\ y \\ z \end{pmatrix} &= \begin{pmatrix} -r\cos\alpha\sin\beta \\ r\cos\alpha\cos\beta \\ 0 \end{pmatrix} \\
\frac{\partial}{\partial \beta}\begin{pmatrix} x \\ y \\ z \end{pmatrix} &= \begin{pmatrix} -r\sin\alpha\cos\beta \\ -r\sin\alpha\sin\beta \\ r\cos\alpha \end{pmatrix}
\end{aligned} \tag{13}$$

Here, we implicitly replaced x, y and z by their corresponding geographic coordinate values using equation (10) before computing the partial derivative. We did not in fact convert an (x, y, z) coordinate vector to an (r, α, β) coordinate vector, so the result is still a vector in Cartesian space. However, each of these vectors have length r, not 1. To turn them into proper unit vectors, we thus need to divide by r. We finally obtain three unit vectors,

$$\mathrm{u}_r = \frac{1}{r}\cdot\frac{\partial}{\partial r}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos\alpha\cos\beta \\ \cos\alpha\sin\beta \\ \sin\alpha \end{pmatrix} \tag{14}$$

which always points outward from the sphere centre,

$$\mathrm{u}_\alpha = \frac{1}{r}\cdot\frac{\partial}{\partial \alpha}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\cos\alpha\sin\beta \\ \cos\alpha\cos\beta \\ 0 \end{pmatrix} \tag{15}$$

which is always tangential to a parallel, and

$$\mathrm{u}_\beta = \frac{1}{r}\cdot\frac{\partial}{\partial \beta}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\sin\alpha\cos\beta \\ -\sin\alpha\sin\beta \\ \cos\alpha \end{pmatrix} \tag{16}$$

which is always tangential to a meridian. Parallels are horizontal circles parallel to the x-y plane, with a position and size defined by the radius and elevation coordinates. Meridians are vertical hemicircles, with a position and size defined by the radius and azimuth coordinates. Most meridians are not parallel to any one of the Cartesian coordinate planes. Rather, they always include both poles of a sphere, and rotate around the axis connecting the poles. See Fig. 11 for an illustration.

B MATLAB code

This section presents the content of the MATLAB file sphereEachLed20160905a.m, including all subfunctions, which we originally used to compute LED positions, create visual stimuli frame by frame, and map one onto the other to generate control signals for each and every one of the LEDs. For a list of contributors, see the preamble of the file itself. The version included here contains more extensive comments than most others, providing a more readily understandable preview of code architecture. It has not been debugged, and is known to contain several errors leading to the local "flipping" of stimuli. These mistakes have been corrected in the alternative version, sphereEachLed20160831k.m. The latter is the version Rebecca Meier used for her work, but it is less extensively commented. It is readily available on demand. Our most up-to-date code combining all sub-functions into one is provided as a separate supplement, S1 Code, published alongside this manuscript.

```
function [ledstatusmap,ledstatus] = sphereEachLed(varargin)
%SPHEREEACHLED Individual LED positions and activity in spherical arena.
%
%   LEDSTATUSMAP = SPHEREEACHLED computes the positions of each individual
%   LED in a spherical stimulus arena (i.e. 64 individual LEDs per LED
%   tile). It then compares these positions to a given stimulus pattern,
%   to provide the on/off status of each individual LED over time.
%
%   LEDSTATUSMAP is a 3D numerical array. Its first two dimensions taken
%   together represent the status of all individual LEDs during one
%   stimulus frame; it is worth noting that the position of LEDs in this
%   two-dimensional matrix is shuffled in a semi-nonsensical way to meet
%   the requirements of the code driving the stimulus arena (see below).
%   The third dimension of LEDSTATUSMAP represents one frame at a time.
%
%   [~,LEDSTATUS] = SPHEREEACHLED returns a 3D numerical array. Its first
%   two dimensions taken together represent the status of all individual
%   LEDs during one stimulus frame. Here, columns represent the 64
%   individual LEDs on one tile, and rows represent one LED from each of
%   the (e.g., 236) tiles. The third dimension of LEDSTATUS represents one
%   stimulus frame at a time.
%
%   There are no required input arguments, but a number of optional ones.
%
%   LEDSTATUSMAP = SPHEREEACHLED(..., 'PARAM1',val1, 'PARAM2',val2, ...)
%   specifies optional parameter name/value pairs to adapt mask shapes and
%   positions, and to control or suppress figure creation. Parameters are:
%
%          'lid' - Is there a lid covering the top or bottom of the sphere?
%                  Valid options are 'none' (default), 'top only' and
%                  'bottom only'.
%     'stimulus' - Custom stimulus pattern. Must be a 2D or 3D numerical
%                  array, where the first dimension represents elevation,
%                  the second represent azimuth, and the third represents
%                  one stimulus frame at a time.
%     'showinfo' - Whether to display debugging messages.
%                  Options are 'yes' (default) and 'no'.
%     'showplot' - Whether to display debugging plots.
%                  Options are 'yes' (default) and 'no'.
%
%   Which of these parameter/value pairs are specified and which ones are
%   left at default values is completely up to the user. Also, they can be
%   specified in any order, as long as they are specified as pairs.
%
%   --------
```

```
%
%   Code example 0: To display this help file from the command window, call
%     help sphereEachLed
%
%   Code example 1: To obtain LED status for the default stimulus, and
%     without a lid covering the sphere arena, call
%     ledstatusmap = sphereEachLed;
%
%   Code example 2: To obtain LED status for a custom stimulus, call
%     ledstatusmap = sphereEachLed('stimulus',mystimulus);
%
%   Code example 3: To obtain LED status for a custom stimulus, while at
%   the same time suppressing plots and debugging messages, call
%     ledstatusmap = sphereEachLed('stimulus',mystimulus,' ...
%                                  'showinfo','no','showplot','no');
%
%   --------
%
%   This function requires other custom functions. These may be included
%   in this file (scroll down to verify), or included as separate .m files.
%   When split into individual .m files, functions should be named thus:
%
%     sphereShape.m             -  computes overall sphere shape
%     sphereTilePosition.m      -  computes LED tile positions
%     sphereLedPosition.m       -  computes individual LED positions
%     spherePlotLedPosition.m   -  optional, shows individual LED positions
%     sphereLedStatus.m         -  computes when each LED must be on or off
%     sphereFakeCylinder.m      -  arranges tiles by ID number to match code
%     spherePlotStatus.m        -  optional, shows individual LED activity
%
%   Version number and credits apply to all of these functions as a whole.
%
%   (In addition, the function STIMULUSBAR is called to create the default
%   stimulus pattern. If a custom pattern is provided to SPHEREEACHLED by
%   specifying the 'pattern' parameter, that pattern is used instead.
%
%     sphereStimulusPattern.m  -  optional, creates default stimulus
%
%   This function STIMULUSBAR is a standalone function and comes with its
%   own documentation. Thus, it is always contained in a separate file.
%   Compatibility has been tested for STIMULUS of version 2016-08-25a.)
%
%   --------
%
%   This is version 2016-09-02e.
%
%   Created by Florian Alexander Dehmelt, U Tuebingen, 7 August 2016.
%   Based on an earlier set of functions by Julian Hinz, U Tuebingen, with
%   contributions by Kun Wang, U Tuebingen. If you require any changes,
%   let me know: florian.dehmelt@uni-tuebingen.de



    % PARSE VARIABLE INPUT ARGUMENTS
    %
    % Check which optional input arguments were provided, and whether their
    % values were provided in the correct format, e.g. a real number, a
    % positive real number, a character array, etc.; if an argument was not
    % provided, assign a default value instead.

    % Note: by default, a standard stimulus pattern is created. If a custom
    % stimulus pattern is provided by specifying the 'pattern' input
    % parameter of SPHEREEACHLED, that custom pattern is used instead.
```

```

close all
p = inputParser;

validstimulus = @(x) isnumeric(x) && numel(size(x))==3;
validshow     = @(x) strcmp(x,'yes') || strcmp(x,'no');
validlid      = @(x) strcmp(x,'none') || ...
                     strcmp(x,'top only') || ...
                     strcmp(x,'bottom only');

default.lid = 'none';
addOptional(p,'lid',default.lid,validlid);

default.stimulus = stimulusBar('barwidth',30,'direction',0, ...
                               'showplot','no','numframe',360);
addOptional(p,'stimulus',default.stimulus,validstimulus);

default.showinfo = 'yes';
addOptional(p,'showinfo',default.showinfo,validshow);

default.showplot = 'yes';
addOptional(p,'showplot',default.showplot,validshow);

parse(p,varargin{:});

lid      = p.Results.lid;
stimulus = p.Results.stimulus;
showinfo = p.Results.showinfo;
showplot = p.Results.showplot;



% DISPLAY WELCOME MESSAGE
%
% Display a welcome message on the command line, containing some basic
% instructions for the user.

if strcmp(showinfo,'yes')
  display(['To display the help file for this function, enter the ', ...
           'following command in the MATLAB command window: ', ...
           'help sphereEachLed'])
end



% COMPUTE SPHERE SHAPE
%
% Compute the overall shape of each hemisphere, its structural ribs, and
% the ribbons of LED tiles in between the ribs. This generates a sphere
% with a default radius, which can be altered inside the function
% sphereShape. Once sharing this code, different radii could be offered.

[sphereradius, ribbonradius, ribbonangle, ribangle] = sphereShape(lid);



% COMPUTE TILE POSITIONS
%
% Manually set the number of tiles per ribbon (top to bottom, meridian to
% side - i.e., decreasing elevation from +90, increasing azimuth from 0),
% for one hemisphere of the spherical arena. This computation could
% easily be automated before sharing the code; as long as it is used in
% conjunction with our arena, there is no need to do so. Afterwards,
```

```
% compute the geographic coordinates of all tile centres.

% Set number of tiles in each ribbon (top to bottom, meridian to side).
switch lid
  case 'none'
    numtile = [5 9 11 13 14 15 14 13 11 8 5];
  case 'top only'
    numtile = [2 5 9 11 13 14 15 14 13 11 8 5];
  case 'bottom only'
    numtile = [5 9 11 13 14 15 14 13 11 8 5 2];
  otherwise
    error(['-- How many lids are covering the poles of the sphere? ', ...
           'Three scenarios are possible: ''none'', ''top only'', ', ...
           'and ''bottom only''. Please select one of them. --']);
end

% Compute the position of LED tiles (holding 64 LEDs each),
% for one hemisphere of the spherical arena.
tilepos = sphereTilePosition(ribbonradius, ribbonangle, ribangle, ...
                             numtile, lid);

% Note: "position" refers to the actual geographic coordinates in actual
% space, not the sequential ID numbers (e.g. 1 to 128) by which the tiles
% are called. These will be set and assigned further below.



% COMPUTE LED POSITIONS
%
% Compute the positions of all individual LEDs (64 per tile),
% for one hemisphere of the spherical arena.

% Given the position of the tiles, and under the assumption that all
% tiles are perpendicular to the sphere surface, compute LED positions
% in both cartesian and geographic coordinates.
[ledposcartesian,ledposgeographic] = ...
  sphereLedPosition(tilepos,sphereradius);

% Replicate the second hemisphere by creating a mirror-symmetric image.
% Do so for individual LED positions expressed both in cartesian...
ledposcartesian2 = ledposcartesian .* ...
                   repmat([1;-1;1], [1, ...
                                   size(ledposcartesian,2), ...
                                   size(ledposcartesian,3), ...
                                   size(ledposcartesian,4)]);

ledposcartesian = cat(4, ledposcartesian, ledposcartesian2);

% ...and in geographic coordinates.
ledposgeographic2 = ledposgeographic .* ...
                    repmat([1;-1], [1, ...
                                   size(ledposgeographic,2), ...
                                   size(ledposgeographic,3), ...
                                   size(ledposgeographic,4)]);

ledposgeographic = cat(4, ledposgeographic, ledposgeographic2);



% DISPLAY LED POSITIONS
%
% Display the position of each individual LED in both cartesian and
% geographic coordinate systems to verify their proper arrangement.
```

```
  % This step is not essential, and should be used for debugging only.

  if strcmp(showplot,'yes')
    spherePlotLedPosition(ledposcartesian,ledposgeographic)
  end



  % COMPUTE LED ACTIVITY STATUS
  %
  % Compare the chosen stimulus pattern to the positions of individual LEDs
  % to find out which one should be active at what time.

  ledstatus = sphereLedStatus(stimulus,ledposgeographic);




  % DISPLAY LED ACTIVITY OVER TIME
  %
  % If plots are desired, show the computed on/off status of each
  % individual LED. This is slow, so it should be used for debugging only.

  if strcmp(showplot,'yes')
    spherePlotStatus(ledstatus,ledposcartesian)
  end



  % REARRANGE LED ACTIVITY MAP TO ACCOMODATE EXISTING CODE
  %
  % As the existing code driving our stimulus arena expects LED tiles to be
  % arranged on the surface of a cylinder (or on a flat rectangular
  % surface), we need to rearrange our spherical distribution of LED tiles
  % into such a rectangular array. The resulting position of certain tiles
  % may seem nonsensical (a tile from the top ending up in the centre of
  % the rectangle, its neighbour up in the bottom right corner etc.), but
  % this new rearrangement is only virtual and has no deeper meaning
  % besides getting the code to work properly. Don't worry about it.

  % Based on the tile ID number displayed by the spherical arena, arrange
  % LED status information in different parts of a rectangular array.
  ledstatusmap = sphereFakeCylinder(ledstatus,lid,showinfo);

end

% The main function ends here. Below are all(!) required custom functions
% called by the main function. These can be moved to appropriately named,
% separate .m files if desired, but doing so may lead to version conflicts.






%% Function to compute the physical parameters of the basic sphere.
function [sphereradius, ...
          ribbonradius, ribbonangle, ribangle] = sphereShape(lid)

  % How many ribbons of LEDs are there, how wide are they, and the
  % structural ribs between them? Because of the inclination of the LED
  % tiles and because of inevitably imperfect spacing, effective ribbon
  % size is larger than originally planned. See note below.
```

```

numribbon = 11;
tilewidth = 21; % Here, used only to compute elevation of ribs/ribbons.
ribwidth = 2.1; % Here, used only to compute elevation of ribs/ribbons.

% NOTE: Elsewhere, the tilewidth is 20 (which is exact). Suggestion:
% Set ribwidth to 3.1 instead to describe the width of the rib itself, as
% well as the width of the inevitable extra space next to it. Then, we
% can use the "true" tilewidth = 20 throughout and avoid confusion.

% To minimize the gap near the poles of the sphere, determine the radius
% for which the sum of the elevation angles covered by LED tiles, covered
% by structural ribs, and covered by the desired holes at the top and
% bottom equals approaches 180 degrees. The following equation is a cost
% function penalising any deviation from this optimal radius.
eqn = @(radius) abs((numribbon)   * 2*asind((tilewidth/2)/radius) + ...
                    (numribbon+1) * 2*asind((ribwidth/2)/radius) + ...
                    1             * 2*asind(60.1/(2*radius)) - 180);

% Numerically solve the equation to find the optimal sphere radius.
sphereradius = fminsearch(eqn,50);

% Next, Julian decided to deviate from the optimum for practical reasons.
% The original code did NOT work for any stretch factors other than 1.05,
% but this problem has been fixed since. All factors >= 1.05 should work.
stretchfactor = 1.05;
sphereradius = sphereradius * stretchfactor;

% Here, "sphereradius" refers to the radius of the sphere on which the
% centres of all LED tiles are located. The following "outerradius" is
% the radius of the sphere on which the inner edges of tiles are located.
outersphereradius = sqrt(sphereradius^2+(tilewidth/2)^2);

% (Re-)Compute the elevation angles covered by a tile, and by a rib.
ribbonangle = 2*asind((tilewidth/2)/sphereradius);
ribangle    = 2*asind((ribwidth/2)/sphereradius);

% Compute the radii of perfectly horizontal planes containing one rib
% each. This computation assumes an odd number of ribbons, i.e. the
% presence of an equatorial ribbon, rather than an equatorial rib.
% Because of symmetry, only the unique radii are computed, then
% replicated once for their mirror-symmetric counterpart.

% First, compute the number of unique ribs.
numuniquerib = (numribbon-1)/2+1;

% Second, compute the unique circular radii.
ribbonradius = NaN(numuniquerib,1);
for k = 1:((numribbon-1)/2+1);
  ribbonradius(k) = outersphereradius * cosd((k-1/2)*ribbonangle ...
                    + (k-1)*ribangle);
end

% Third, replicate the mirror-symmetric copies.
ribbonradius = [fliplr(ribbonradius(2:numuniquerib)'), ribbonradius'];

% Fourth, if a lid is present, add an additional circular radius.
if ~strcmp(lid,'none')

  lidradius = outersphereradius * ...
              cosd((numuniquerib+.5)*ribbonangle + numuniquerib*ribangle);

  % Depending on where the lid is located, place its radius
```

```
      % at the top or at the bottom of the list of circular radii.
      switch lid
        case 'top only'
          ribbonradius = [lidradius ribbonradius];
        case 'bottom only'
          ribbonradius = [ribbonradius lidradius];
        otherwise
          error(['-- How many lids are covering the poles of the ', ...
                 'sphere? Three scenarios are possible: ''none'', ', ...
                 '''top only'', and ''bottom only''. Please select ', ...
                 'one of them.--']);
      end

    end

  end





  %% Function to compute position of LED tiles from basic sphere shape.
  function tilepos = sphereTilePosition(ribbonradius, ribbonangle, ...
                                       ribangle, numtileperribbon, lid)

    ribbonbeta = (ribbonangle+ribangle) * (5:-1:-5);

    % To create the correct number of ribbons, check whether lid is present.
    switch lid
      case 'none'
        % Relax. Do nothing.
      case 'top'
        ribbonbeta = [ribbonangle*6 + ribangle*6, ribbonbeta];
      case 'bottom'
        ribbonbeta = [ribbonbeta, -(ribbonangle*6 + ribangle*6)];
      otherwise
        error(['-- How many lids are covering the poles of the sphere? ', ...
               'Three scenarios are possible: ''none'', ''top only'', ', ...
               'and ''bottom only''. Please select one of them. --']);
    end

    % Preallocate variable size.
    tilepos = NaN(sum(numtileperribbon),5);

    % In principle, some tiles may be flipped upside down, while others are
    % not. To accomodate for this, we will be keeping track of a variable
    % indicating whether a specific tile is flipped or not. For now, all
    % tiles will be set to 1 ("flipped"), rather than 0 ("upright"), because
    % this is the case in our current setup.
    tilepos(:,1) = 1;

    % Run counter for each iteration to save tile values in different places.
    counter = 0;

    % Go through all rows...
    for a = 1:length(ribbonradius)

      % ...and within each row, go through all of its elements.
      for b = 1:numtileperribbon(a)

        % Compute the azimuth of each tile centre,
        % taking into account the 6mm wide meridian "keel" or "spine",
        % and the 20mm width of each tile.
```

```
      keelwidth = 6;
      tilewidth = 20;

      % Compute the azimuth covered by the keel, and by each tile.
      % These numbers are exact for the keel, as well as for tiles along
      % the equatorial ribbon. For all other ribbons, they are ONLY
      % APPROXIMATE, because these tiles are perpendicular to the
      % sphere, but not perpendicular to the circle formed by the ribbon
      % (i.e., they are not perfectly vertical). Their true azimuth spread
      % would be slightly larger.
      keelangle = 2*asind((keelwidth/2)/ribbonradius(a));
      tileangle = 2*asind((tilewidth/2)/ribbonradius(a));

      % Compute the azimuth of the centre of this tile,
      % considering the azimuth offset and the tiles already placed.
      thistilealpha = keelangle + (b-1/2)*tileangle;

      % Counter to save in the desired order.
      counter = counter + 1;
      tilepos(counter,3) = thistilealpha;
      tilepos(counter,4) = ribbonbeta(a);
      tilepos(counter,5) = ribbonradius(a);

    end

  end

end





%% Function to compute individual LED positions from tile positions.
function [ledposcartesian, ...
          ledposgeographic] = sphereLedPosition(tilepos,sphereradius)

  ledseparation = 2.48;
  numtile = size(tilepos,1);
  tileisflipped = tilepos(:,1);

  % Pre-allocate variable size to speed up computation.
  ledpos           = NaN(length(tilepos)*64,5);
  ledposcartesian  = NaN(3,8,8,numtile);
  ledposgeographic = NaN(2,8,8,numtile);

  for tile = 1:numtile

    % Read out the position of the tile centre in geographic coordinates.
    tilealpha  = tilepos(tile,3);
    tilebeta   = tilepos(tile,4);
    tileradius = sphereradius;

    % Convert the position of the tile centre into cartesian coordinates.
    tilecentre = tileradius * [cosd(tilebeta)*cosd(tilealpha); ...
                               cosd(tilebeta)*sind(tilealpha); ...
                               sind(tilebeta)];

    % Compute the "unit" vector in the beta direction (still cartesian).
    betaunitvector = [-sind(tilebeta)*cosd(tilealpha); ...
                      -sind(tilebeta)*sind(tilealpha); ...
                       cosd(tilebeta)];

```

```
    % Normalise the vector to make sure it is a unit vector.
    betaunitvector = betaunitvector/norm(betaunitvector);

    % Compute the "unit" vector in the alpha direction  (still cartesian).
    alphaunitvector = [-cosd(tilebeta)*sind(tilealpha); ...
                       cosd(tilebeta)*cosd(tilealpha); ...
                       0];

    % Normalise the vector to make sure it is a unit vector.
    alphaunitvector = alphaunitvector/norm(alphaunitvector);


    % Now, place 64 individual LEDs around the tile centre.
    for row = 1:8    % go through columns (!)
      for col = 1:8  % go through rows (!)

        % Compute LED position in cartesian coordinates.
        b = ledseparation * (row - 4.5);
        a = ledseparation * (col - 4.5);
        rled = tilecentre + b*betaunitvector + a*alphaunitvector;

        % Save them for later.
        ledposcartesian(:,row,col,tile) = rled;

        % Convert LED position from cartesian to geographic coordinates.
        beta  = asind(rled(3)/norm(rled));
        alpha = atand(rled(2)/rled(1));

        % Constrain geographic coordinate values to standard range.
        % beta = mod(beta,180)-90; % Not needed.
        alpha = mod(alpha,180);

        % Save them for later.
        ledposgeographic(:,row,col,tile) = [beta,alpha];

        % Remember which tile this LED is on (i.e., its ID number).
        id = (tile-1)*64 + (row-1)*8 + col;

        ledpos(id,2) = tilepos(tile,2); % Remember the tile ID number.

      end
    end
  end

  allisflipped = floor(sum(tileisflipped)/numel(tileisflipped));

  if allisflipped  % All tiles were flipped.

    % Were all tiles accidentally flipped upside down during construction?
    % If so, flip them back the way they belong.
    ledposcartesian = flipdim(flipdim(ledposcartesian, 2), 3);
    ledposgeographic = flipdim(flipdim(ledposgeographic, 2), 3);

  elseif sum(tileisflipped)  % At least some tiles were flipped.

    % Were individual tile flipped upside down during construction?
    % Or, more precisely, rotated 180 degrees around its centre point?
    % If so, flip the LED coordinates back the way they belong.

    for tile = 1:numtile
      if tileisflipped(tile)

        ledposcartesian(:,:,:,tile) = ...
```

```
        flipdim(flipdim(ledposcartesian(:,:,:,tile), 2), 3);

      ledposgeographic(:,:,:,tile) = ...
        flipdim(flipdim(ledposgeographic(:,:,:,tile), 2), 3);

    end
  end

 end

end




%% Function to assign custom tile numbers to tile positions.
function rearranged = sphereFakeCylinder(original,lid,showinfo)

 % FIRST, REORDER TILES BY TILE ID NUMBER
 %
 % Rearrange information on the activity of each individual LED based on
 % the ID number of the tile upon which they are located. These tile
 % numbers (ranging from 1 to 240) are displayed on the spherical arena.

 % List the ID numbers of all tiles in one hemisphere, top-to-bottom and
 % meridian-to-side (i.e., going through one horiz. row after another).
 hemisphere1 = ...
   [115, 116,  65,  45,  47, ...
    113, 114,  80,  78,  66,  46, 48,  4,  2, ...
    119, 120, 118,  79,  77,  67, 56, 15, 14, 16,  3, ...
    112, 111, 110, 109, 117,  61, 68, 54, 38, 37, 12, 13,  1, ...
    108, 107, 106, 105,  85,  86, 63, 52, 55, 42, 39, 11, 24,  6, ...
     84,  83,  82,  81,  88,  87, 64, 51, 53, 43, 40,  9, 10, 23, 5, ...
    104, 103, 102, 101,  74,  76, 62, 50, 44, 41, 25, 18, 17, 22, ...
    100,  98,  97,  90,  73,  75, 49, 28, 27, 26, 20, 19, 21, ...
     99,  92,  89,  72,  71,  60, 58, 33, 34, 30, ...
     29,  94,  91,  70,  69,  59, 36, 35, 31, ...
     95,  96,  93,  57,  32];

 % List the ID numbers of tiles on the first-hemisphere side of the lid.
 lid1 = [7, 8];

 % List the ID numbers of the tiles on the second hemisphere. The order is
 % top-to-bottom and meridian-to-side again - so it is mirror-symmetric to
 % the order of the first hemisphere (the actual ID numbers assigned to
 % each tile can be arbitrarily different, though).
% hemisphere2 = ...
%   [, ...
%     , ...
%     , ...
%     , ...
%     , ...
%     , ...
%     , ...
%     , ...
%     , ...
%     , ...
%     ];

 % Finally, list the ID numbers for the second half of the lid.
% lid2 = [127, 128];
```

```
%   % The following are dummy IDs created for the second hemisphere and the
%   % second half of the lid. They must be replaced with the true IDs there
%   % as soon as Kun Wang has made these available.
  hemisphere2 = hemisphere1 + 120;
  lid2        = lid1        + 120;

  % Aggregate the tile ID numbers of all tiles in the spherical arena in
  % the correct order, taking into account where exactly the lid is placed.
  switch lid
    case 'none'
      neworder = [hemisphere1,hemisphere2];
    case 'top only'
      neworder = [lid1,hemisphere1,lid2,hemisphere2];
    case 'bottom only'
      neworder = [hemisphere1,lid1,hemisphere2,lid2];
  end

  % The following line ensures that unassigned tile numbers are padded with
  % NaNs. If you remove the line, they will be padded with zeros instead -
  % or skipped entirely if there are no higher, actually assigned numbers.
  % To safely pad the array with zeros, replace NaN(...) with zeros(...).
  numframe = size(original,3);
  reordered = NaN(8,8,numframe,240);

  % Rearrange the fourth dimension of the arrays containing LED positions.
  % Remember that dimension 1 are the actual coordinates (e.g., azimuth and
  % elevation), dimensions 2 and 3 cluster the 8*8 individual LED on each
  % tile, and dimension 4 goes through all tiles in the setup. The old
  % order went through all tiles top-to-bottom, meridian-to-side; the new
  % order goes through all tiles from the tile with ID no. 1 to the tile
  % with ID no. 236 (or whatever else the maximum is).
  numtile = numel(neworder);
  oldorder = 1:numtile;
  reordered(:,:,:,neworder) = original(:,:,:,oldorder);

  % For debugging (and only for debugging), display how many tile IDs were
  % found, and how many more could be used.
  if strcmp(showinfo,'yes')
    unassigned = numel(find(isnan(reordered)))/(64*size(original,1));
    display(['-- Out of 240 supported tiles, ',num2str(numtile), ...
             ' tile IDs were assigned; ',num2str(unassigned), ...
             ' were left unassigned. --'])
  end



  % SECOND, REARRANGE TILES INTO A VIRTUAL, RECTANGULAR PATTERN
  %
  % As the existing code driving our stimulus arena expects LED tiles to be
  % arranged on the surface of a cylinder (or on a flat rectangular
  % surface), we need to rearrange our distribution of LED tiles into such
  % a rectangular array. The resulting position of certain tiles may seem
  % nonsensical (a tile from the top ending up in the centre of the
  % rectangle, its neighbour up in the bottom right corner etc.), but this
  % new rearrangement is only virtual and has no deeper meaning besides
  % getting the code to work properly. Don't worry too much about it.

  % Arrange increasing tile ID top-to-bottom, then left-to-right, in
  % vertical columns of 8. The total number of columns is 15 for 120 tiles,
  % 30 for 240 tiles.

  rearranged = NaN(64,240,numframe);

```

```
%   % The following is a HACK. Clean up in the near future. (!!!)
%   reordered = permute(reordered,[2 1 3 4]);

  for tileID = 1:numtile

    xshift = 8*floor((tileID-1)/8);
    yshift = 8*mod(tileID-1,8);

    rearranged((1:8)+56-yshift,(1:8)+xshift,:) = reordered(:,:,:,tileID);

  end


end





%% Function to display the positions of all individual LEDs.
function spherePlotLedPosition(ledposcartesian,ledposgeographic)

  % Part 1/2: Plot LED positions in cartesian coordinates.
  figure(44)
  set(gcf,'Color',[1 1 1])

  % Read out the position data.
  rledplot = reshape(ledposcartesian,[3 numel(ledposcartesian)/3]);
  numtile = size(ledposcartesian,4);

  % Divide the tiles into four groups, to be assigned one of four colours.
  group1 = 1:4:numtile;
  group2 = 2:4:numtile;
  group3 = 3:4:numtile;
  group4 = 4:4:numtile;

  % Find all the individual LEDs belonging to each group of tiles.
  index1 = repmat(1:64,[1 numel(group1)]) + ...
          reshape(64*ones(64,1)*(group1-1), [1 64*numel(group1)]);
  index2 = repmat(1:64,[1 numel(group2)]) + ...
          reshape(64*ones(64,1)*(group2-1), [1 64*numel(group2)]);
  index3 = repmat(1:64,[1 numel(group3)]) + ...
          reshape(64*ones(64,1)*(group3-1), [1 64*numel(group3)]);
  index4 = repmat(1:64,[1 numel(group4)]) + ...
          reshape(64*ones(64,1)*(group4-1), [1 64*numel(group4)]);

  % Plot the individual LEDs, one group after another.
  hold on
  s1 = scatter3(rledplot(1,index1),rledplot(2,index1),rledplot(3,index1));
  s2 = scatter3(rledplot(1,index2),rledplot(2,index2),rledplot(3,index2));
  s3 = scatter3(rledplot(1,index3),rledplot(2,index3),rledplot(3,index3));
  s4 = scatter3(rledplot(1,index4),rledplot(2,index4),rledplot(3,index4));
  hold off

  % Adjust LED plot size, and colour them in their group's colour.
  axis equal
  colour = [[.8 .4 .2]; [.2 .4 .8]; [.2 .8 .6]; .2*[1 1 1]];
  set(s1,'MarkerEdgeColor',colour(1,:),'MarkerFaceColor',colour(1,:))
  set(s2,'MarkerEdgeColor',colour(2,:),'MarkerFaceColor',colour(2,:))
  set(s3,'MarkerEdgeColor',colour(3,:),'MarkerFaceColor',colour(3,:))
  set(s4,'MarkerEdgeColor',colour(4,:),'MarkerFaceColor',colour(4,:))
  set([s1,s2,s3,s4],'SizeData',2)

```

```
  % Part 2/2: Plot LED positions in geographic coordinates.
  figure(45)
  set(gcf,'Color',[1 1 1])
  axis([-180 180 -90 90])
  box on
  xlabel('Azimuth')
  ylabel('Elevation')
  plot(ledposgeographic(2,:), ledposgeographic(1,:),'ko','MarkerSize', 2)

end





%% Function to compute when each LED should be on or off.
function ledstatus = sphereLedStatus(pattern,ledposgeographic)

  % Create a discrete grid in geographic coordinates. This grid must have
  % the same resolution as the stimulus pattern.
  betagridstep  = 180/(size(pattern,1)-1);
  alphagridstep = 360/(size(pattern,2)-1);
  betagrid  = (-90:betagridstep:+90)';
  alphagrid = (-180:alphagridstep:+180)';

  % How many individual LEDs are on each tile, how many tiles are there?
  numtile = 240;
  numframe = size(pattern,3);
  ledstatus = NaN(8,8,numframe,numtile);

  % Go through every single LED on every single tile, by row and column.
  for tile = 1:size(ledposgeographic,4)
    for row = 1:size(ledposgeographic,3)
      for col = 1:size(ledposgeographic,2)

        % Find the geographic grid point closest to the exact LED position.
        beta  = ledposgeographic(1,row,col,tile);
        alpha = ledposgeographic(2,row,col,tile);
        [~ , bestbeta]  = min(abs(beta-betagrid));
        [~ , bestalpha] = min(abs(alpha-alphagrid));

        % The LED status is the value of this grid point.
        ledstatus(row,col,:,tile) = pattern(bestbeta,bestalpha,:);

%         % The following is a HACK. Clean up in the near future. (!!!)
%         % (Not sure why it's not "row,col" instead...)
%         ledstatus(col,row,:,tile) = pattern(bestbeta,bestalpha,:);


      end
    end
  end

end





%% Function to display the activity of all individual LEDs (optional)
function spherePlotStatus(ledstatus,ledposcartesian)

  figure(46)
```

```
  set(gcf,'Color',[1 1 1],'Position',[200 0 700 700])

  % Find the x, y and z coordinates of all LEDs, write them as a 3D array.
  % This array is row x column x tile ID, i.e., 8 x 8 x number of tiles.
  xled = squeeze(ledposcartesian(1,:,:,:));
  yled = squeeze(ledposcartesian(2,:,:,:));
  zled = squeeze(ledposcartesian(3,:,:,:));

  % Preallocate variable size, compute coordinates of a ball (see below).
  numframe = size(ledstatus,3);
  [xball,yball,zball] = sphere;
  ballradius = .95 * norm(ledposcartesian(:,1,1,1));

  % Go through one frame after another, creating a video of the stimulus.
  for k = 1:numframe

    % Find out which LEDs are on or off on this frame.
    ongroup  = squeeze(ledstatus(:,:,k,:)==1);
    offgroup = squeeze(ledstatus(:,:,k,:)==0);

    % Display the active LEDs on this frame.
    scatter3(xled(ongroup),yled(ongroup),zled(ongroup), ...
            'MarkerFaceColor',[.4 1 .6], ...
            'MarkerEdgeColor',[.2 .2 .2], ...
            'SizeData',30)
    set(gca,'XLim',[-120 120],'YLim',[-120 120],'ZLim',[-120 120])

    hold on

    % Optionally, plot the inactive LEDs as well (slowing it all down).
%     scatter3(xled(offgroup),yled(offgroup),zled(offgroup), ...
%             'MarkerFaceColor',[.2 .2 .2], ...
%             'MarkerEdgeColor',[.2 .2 .2], ...
%             'SizeData',35)
%     set(gca,'XLim',[-120 120],'YLim',[-120 120],'ZLim',[-120 120])

    % Add a partly transparent ball to facilitate depth perception.
    surf(ballradius*xball,ballradius*yball,ballradius*zball, ...
        'EdgeColor','none','FaceColor',[1 1 1],'FaceAlpha',.4)
    text(0,550,['frame ',num2str(k)],'FontSize',32,'Units','pixels')
    hold off

    % Add a pause to force MATLAB to display the movie more smoothly.
    pause(.01)

  end

  % Ensure that all axes are displayed with proper scaling, so the sphere
  % really appears as a sphere, not a distorted ellipsoid.
  axis square

end
```