Junior-Prof. Dr. Olaf Ronneberger,
Image Analysis Lab, Institute of Computer Science, Albert-Ludwigs-University Freiburg

Exercises for 3D Image Analysis

Summer term 2015

Exercise 2 (Issue Date: Fri, 08.05.2015, Due Date: Fri, 22.05.2015, **1pm**)

1) Preliminaries – Set up Blitz++ library

1) Install the Blitz++ library on your machine.

   For this you have 2 options (for both you need to be root):

   – Install the standard package (recommended):

   ```
   sudo apt-get install libblitz0-dev
   ```

   – Download the latest version from the below link, and install it according to the documentation:

   http://sourceforge.net/projects/blitz/

   Note: It should be already installed on the pool machines!

2) Write a simple test program using blitz::Array and compile it. If the library is installed correctly, you shouldn't need any particular arguments for compilation and the following command:

   ```
   g++ -Wall -g test.cc -o test -lblitz
   ```

   Should do. However if it complains about not finding the included files or libraries, the following arguments can be added as follows (default paths are used in this example):

   ```
   g++ -Wall -g test.cc -o test -I/usr/include -L/usr/lib -lblitz
   ```

   Using Blitz++ library, 3D arrays (or multi-dimensional arrays in general) can be handled very comfortably. Use the example code “test.cc” to generate a 3D blitz++ array and to write it in raw format.

3) Install ImageJ from http://rsbweb.nih.gov/ij/ to load and view your created data set

   When importing raw data in ImageJ, the data extents have to be specified. Use the information given in the example code, or filename “test_41x51x61_8bit.raw” respectively.

2) Movie of Rotating Pollen Grain and Zebrafish

The goal of this exercise is to render a movie of a rotating pollen grain (or Zebrafish, which is more impressive) from a given 3D volumetric data set. The datasets were recorded with a confocal laser scanning microscope. The voxel sizes of the given datasets are:

Pollen grain:	0.4µm x 0.2µm x 0.2µm
Zebrafish:	3.9µm x 1.49155µm x 1.49155µm

Create a single source file named "render_movie.cc", that contains all the code for this part of the exercise.

1) We want to use blitz::TinyVector and the blitz::TinyMatrix classes. The usage of the provided method of the blitz++ library for Matrix products (blitz::product) does not work as expected in many situations (due to its internal optimization with iterators). Therefore, write your own implementations:

- blitz::TinyMatrix<float,4,4> **myproduct**(blitz::TinyMatrix<float,4,4> **a**, blitz::TinyMatrix<float,4,4> **b**);
- blitz::TinyVector<float,4> **myproduct**(blitz::TinyMatrix<float,4,4> **m**, blitz::TinyVector<float,4> **v**);

2) To retrieve a grayvalue at a real-valued position from an Array usually interpolation is needed. We will implement different interpolation schemes in later exercises. For now it is enough to use a simple nearest neighbour interpolator. If the requested position is outside the Array bounds, it shall return 0

- unsigned char **interpolNN**(const blitz::Array<unsigned char, 3>& **arr**, blitz::TinyVector<float,4> **pos**);

3) To obtain a dense output dataset, we use the inverse transformation and iterate over the target array positions. To speed up the transformation all necessary coordinate transformations shall be performed with one single matrix, which transforms a target position (given in array coordinates level, row, column) to the corresponding source position (in array coordinates level, row, column). Write a function

- void **transformArray**(const blitz::Array<unsigned char, 3>& **srcArr,**
  const blitz::TinyMatrix<float,4,4>& **invMat,**
  blitz::Array<unsigend char, 3>& **trgArr**);

The central part of this function could look like this:

```
trgPos = lev, row, col, 1;
srcPos = myproduct( invMat, trgPos);
trgArr(lev,row,col) = interpolNN( srcArr, srcPos);
```

4) The transformation matrix shall describe a rotation around the center of the array, such that the central voxel of source array is projected to the central voxel of the target array (and vice versa). The voxel sizes of volumetric data are usually given in micrometer for each dimension (lev,row,col). Consider specific voxel sizes (element_size_um) for the source and the target array by performing the necessary scaling operations. Write a function

- blitz::TinyMatrix<float,4,4> **createInverseRotationMatrix**(
  const blitz::TinyVector<size_t, 3>& **srcArrShape**,
  const blitz::TinyVector<size_t, 3>& **trgArrShape**,
  const blitz::TinyVector<float,3>& **src_element_size_um**,

```
const blitz::TinyVector<float,3>& trg_element_size_um,
float angleAroundLev,
float angleAroundRow,
float angleAroundCol);
```

To make the function as readable as possible, create seven 4x4 matrices -- one for each sub step, e.g. shiftTrgCenterToOrigin, scaleToMicrometer, rotateAroundLev, rotateAroundRow, rotateAroundCol, scaleToVoxel, shiftOriginToSrcCenter and combine them at the end to one matrix. Attention the transformation that should be applied first, has to be the rightmost matrix in the product of matrices.

5) write a function for saving a pgm image

- void **savePGMImage**(const blitz::Array<unsigned char,2>& **image**,
  const std::string& **fileName**);

6) Write the main() function performing the following tasks:

1. Load the data set "Artemisia_pollen_71x136x136_8bit.raw", or "Zebrafish_71x361x604_8bit.raw" respectively.
2. Create the target Array (must be larger than the source Array, such that the 45 degree rotated array fits into it)
3. for each *angle* from 0 to 355 in 5 degree steps
   - compute the rotated array (rotation around the row-Axis (pollen grain), or col-Axis (Zebrafish) respectively)
   - compute the maximum intensity projection of the rotated Array
   - save the resulting image to "movie_frame_XXX.pgm", where XXX is the (zeropadded) angle

use ImageJ to load the movie "File--Import--Image Sequence" and play it with "Image--Stacks--Start Animation". If you like, you can save it as *.avi using "File--Save As--AVI...", and play it with any video player software. The framerate can be adjusted at "Image--Stacks--Animation Options..." before saving.

3) Anaglyphs

Now we want to render a stereo movie (also called “anaglyph” movie) instead (red-cyan color overlay). You can watch this movie using red-cyan glasses to get a 3D impression. Write a new program for this purpose.

Create a single source file named “render_anaglyph_movie.cc”, that contains all the code for this part of the exercise.

7) write a function for saving a ppm image

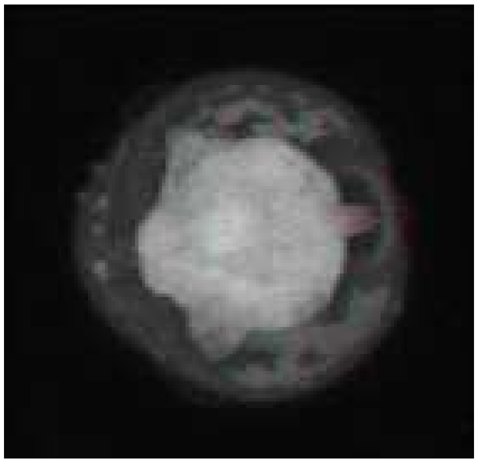
- void **savePPMImage**(
  const blitz::Array<blitz::TinyVector<unsigned char,3>,2>& **image**,
  const std::string& **fileName**);

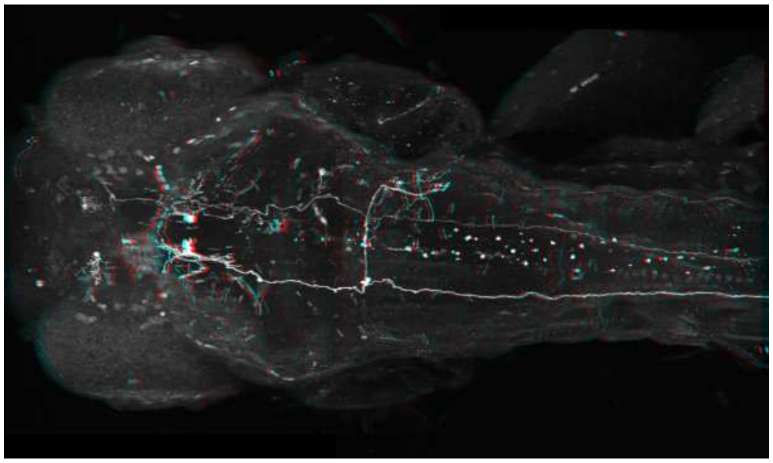
that saves a color image in raw ppm (portable pixmap) format. The ppm format is very similar to the pgm format (see "man ppm" on a linux box). The RGB color values for each pixel are stored in blitz::TinyVectors of dimension 3. A color image is represented as a blitz-array of TinyVectors. A blitz-Array of TinyVectors contains the raw data in the correct order, so you can directly save the memory starting at image.dataFirst().

Anaglyph Image

8) At first, we generate a single anaglyph image. A virtual 3D object can be generated by offering two different views from slightly different angles to the viewer, see Figure 1. This can be done by e.g. encoding the two different views in the two colors red and cyan in a color-overlay. The right and the left eye now have to receive the corresponding image. For this purpose red-cyan glasses are use. The right/cyan glass only transmits the cyan (green+blue) light, while the left/red glass only transmits the red light. This way the left eye only receives the red channel information, which is a view on the object slightly rotated to the right and the right eye only receives the cyan channel information, which is a view on the object slightly rotated to the left, see Figure 2.
See http://en.wikipedia.org/wiki/Anaglyph_image or http://www.3d-foto-shop.de/service/ for some examples and further information.

The generated images should look like the following:


Pollen grain


Zebrafish

Write the main() function performing the following tasks:

1. Load the data set "Artemisia_pollen_71x136x136_8bit.raw", or "Zebrafish_71x361x604_8bit.raw" respectively.
2. Create the target Array (must be larger than the source Array, such that the rotated array fits into it)
3. Create the two different views
   - compute the rotated array (rotation around the row-Axis by +2.5 degree, rotates left)
   - compute the maximum intensity projection of the rotated Array (rightView)
   - compute the rotated array (rotation around the row-Axis by -2.5 degree, rotates right)
   - compute the maximum intensity projection of the rotated Array (leftView)

   Create the color-overlay and save it as a ppm image
   - compute the red-cyan color overlay anaglyphImage (from rightView and leftView)

   ```
   blitz::Array< blitz::TinyVector<unsigned char,3> ,2> anaglyphImage(...)
   ```

   Hint: blitz-Arrays overload the operator[] for comfortable access of the TinyVector components (see chapter 5.1 in the Blitz++ User Guide at http://blitz.sourceforge.net/resources/blitz-0.9.pdf or at http://dsec.pku.edu.cn/~mendl/blitz/manual/blitz05.html#l97). E.g., to access the zero component of all TinyVectors in an vector-array as scalar array (which correspond to the red channel in an RGB image), e.g. just use:

```
blitz::Array<blitz::TinyVector<unsigned char,3>,2> rgbImage( 480, 640);
blitz::Array<unsigned char,2> redChannel( 480, 640);
...
rgbImage[0] = redChannel;
```

  - save the resulting image to "anaglyphImage.ppm"

use ImageJ to load the image and use red-cyan glasses to get a 3D impression.

Anaglyph Movie

Now we want to render a complete anaglyph movie. For this purpose the data set is rotated around the col-Axis, while the local rotation around the row-Axis provides the different views for the anaglyph image as described in sub task 8.

9) add the following workflow to create the anaglyph movie:

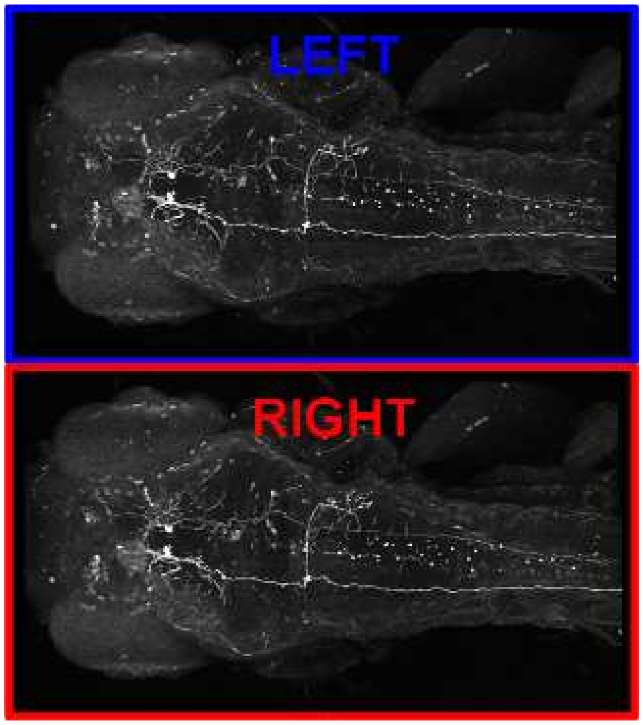
- Create the target Array (must be larger than the source Array, such that the rotated array fits into it – both 45 degree rotations around col-Axis and the rotation around row-Axis have to be considered)
- for each *angle* from 0 to 355 in 5 degree steps (rotation around col-Axis)
  - create the two different views
    - compute the rotated array (rotation around col-Axis by *angle* and row-Axis by +2.5 degree)
    - compute the maximum intensity projection of the rotated Array (rightView)
    - compute the rotated array (rotation around col-Axis by *angle* and row-Axis by -2.5 degree)
    - compute the maximum intensity projection of the rotated Array (leftView)
  - create the color-overlay and save it as ppm image
    - compute the red-cyan color overlay anaglyphImage (from rightView and leftView)
    - save the resulting image to "movie_anaglyph_frame_XXX.ppm", where XXX is the (zeropadded) angle

use ImageJ to load the movie and use red-cyan glasses to get a 3D impression.

4) 3D Movie

Today's 3D cinema movies are not displayed using red-cyan color-overlay. Instead polarized glasses (passive) or active shutter glasses are used to switch between the content for the left and right eye. Now we want to encode the created left and right view (from sub task 9) into a 3D video format that today's 3D video players can handle and display in different 3D video formats (including red-cyan and OpenGL stereo).

We use a very simple format, that stores **over/under-formatted** frames. The left and right frame are stored above each other in a frame with double height:



Create a single source file named "render_3d_movie.cc" (you can copy your solution from "render_anaglyph.cc"), that contains all the code for this part of the exercise.

10) replace the last step from sub task 9 (create the color-overlay and save it as ppm image) by:

- create the over/under-formatted frame and save it as ppm image
  - compute the over/under-formatted frame (from leftView and rightView), the over/under-formatted frame should have a total size of 640x720px, leftView and rightView should be centered into a half frame of size 640x360px each, the left view should be the top and the right view the bottom panel of the over/under-formatted frame (see figure above)
  - save the resulting image to "movie_3d_frame_XXX.ppm", where XXX is the (zeropadded) angle

use ImageJ to load the movie and store it as uncompressed AVI.

You can download the free 3d video player "Bino" from http://bino3d.org/ (available for Linux, Mac and Windows). It allows to play 3d videos from different input formats (including top/bottom format) and to display them in different output formats (including red-cyan, OpenGL stereo). You can view the red-cyan output using red-cyan glasses. (Hopefully we can also use our 3D beamer and shutter glasses to view the OpenGL stereo formated output.)

If you are interested you can visit http://www.tomshardware.com/reviews/blu-ray-3d-3d-video-3d-tv,2632.html to get a nice overview of 3D video techniques and formats.

Youtube also provides video in 3D formats, including red-cyan or over-under format:
http://www.youtube.com/results?search_query=yt3d

Hints:

- To access the raw data of a blitz::Array you can use the data() method. E.g. for loading a dataset you can simply write:

```
inFile.read( reinterpret_cast<char*>( arr.dataFirst()), arr.size()*sizeof(unsigned char));
```

- use the shape() method to get all extents of an array, e.g. createRotationMatrix(srcArr.shape(), ...)
- to create the appropriate filename use a stringstream and iomanipulators, e.g.

```
#include <sstream>
#include <iomanip>
....
std::ostringstream os;
os << "movie_frame_" << std::setw(3) << std::setfill('0') << angle << ".pgm";
std::string fileName = os.str();
```

- to calculate with TinyVectors, e.g. to use blitz::floor(pos + 0.5), where pos is a TinyVector, you have to include the following for blitz version 0.9:

```
#include <blitz/tinyvec-et.h>
```

In blitz-0.10 including only <blitz/array.h> is sufficient.

Appendix:

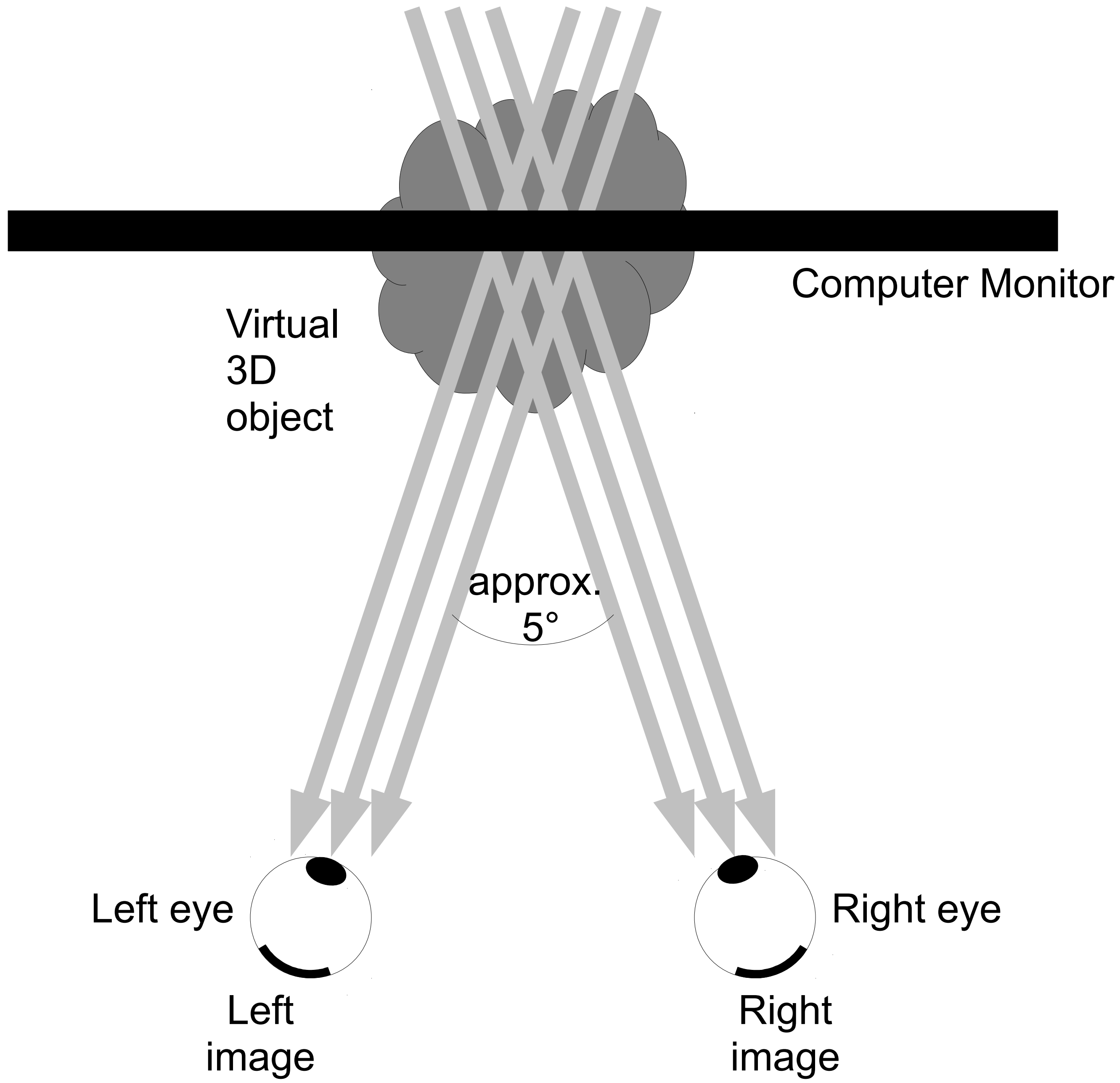


Figure 1: Virtual 3D object

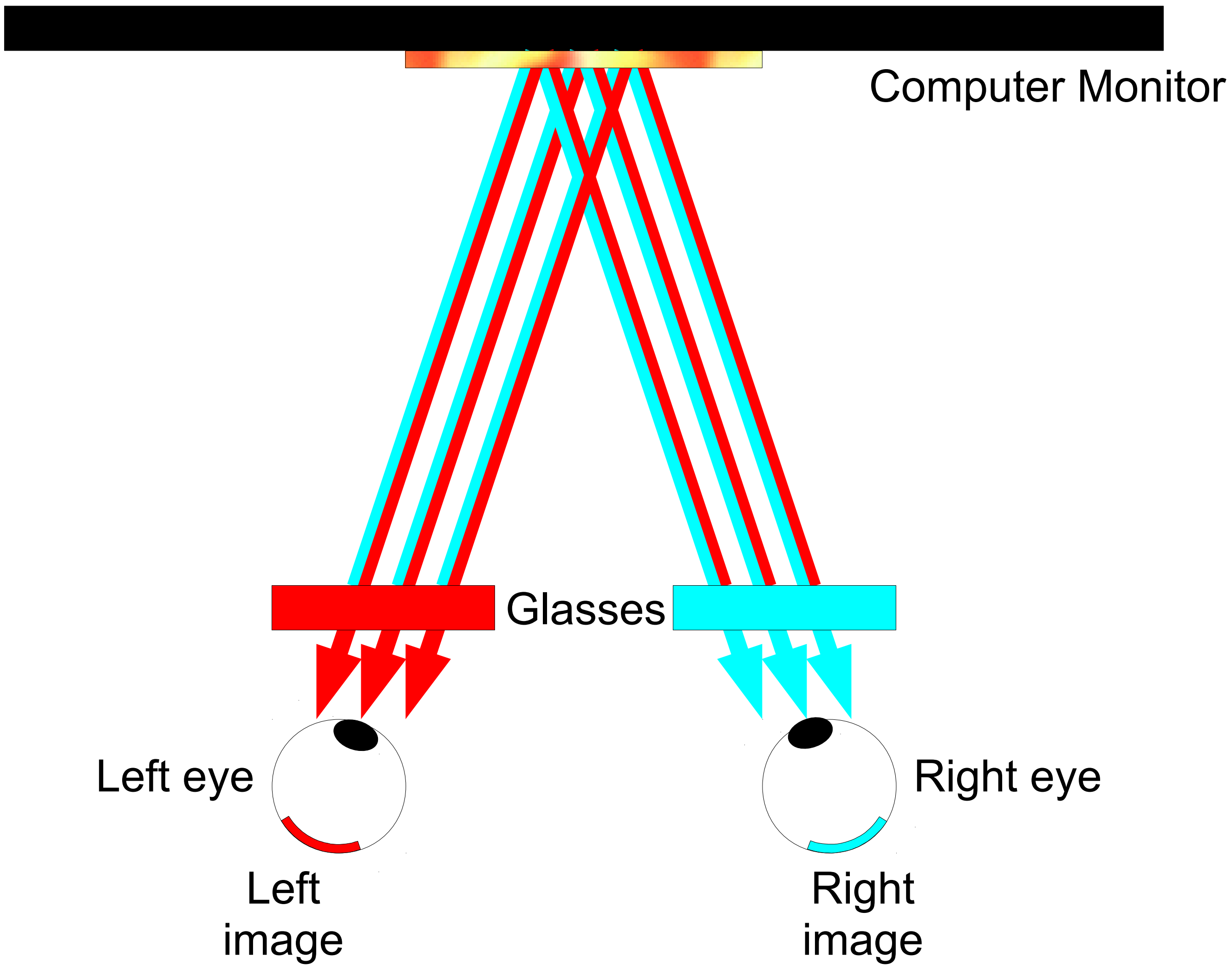


Figure 2: Virtual 3D object by anaglyph image