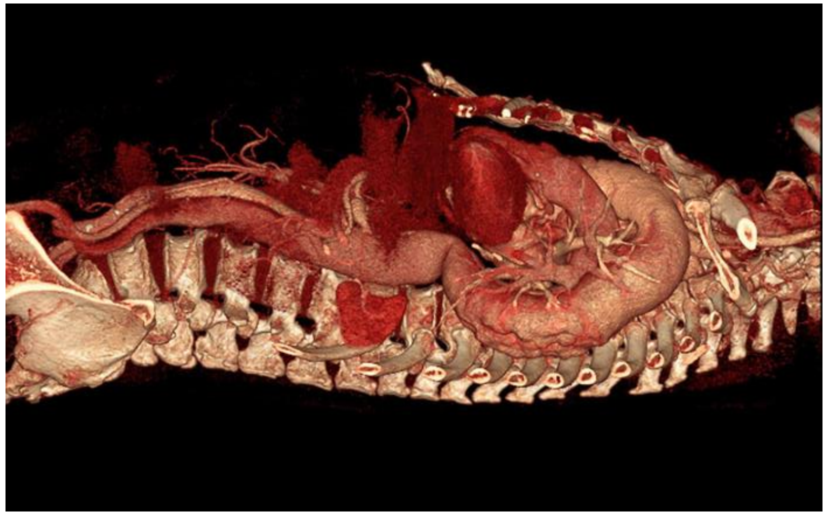
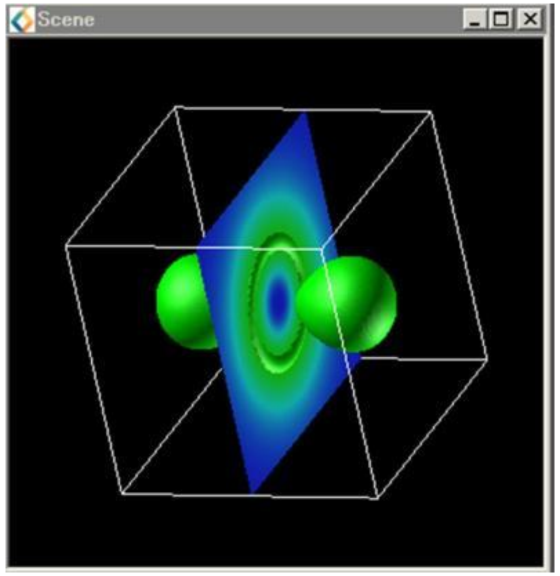
Volume Data

- 3D array of data. Each volume element (voxel) is a value representing some measurement or calculation
- e.g. Magnetic resonance imaging (MRI) data: Strong magnetic field aligns magnetization of hydrogen atoms, and then measures radio waves they emit (body tissue contains a lot of hydrogen (water)). Computed Tomography (CT) data: Interior images of the body are calculated using many X-rays of the body.

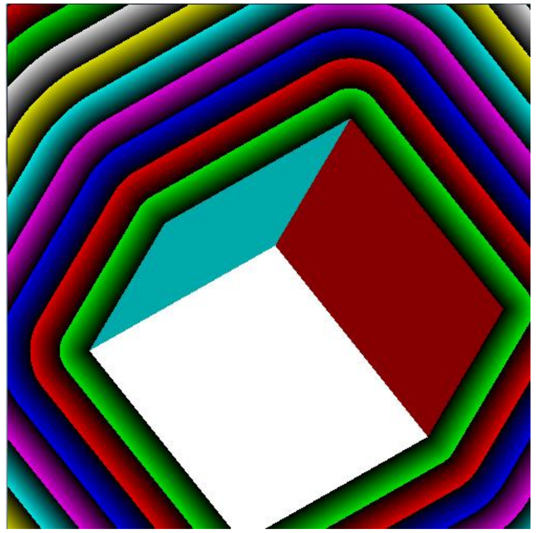


Volume Data

- e.g. simulation: probability of electron position around a H_2 molecule
- Machine representation simple: 3D array of char/integer/real, etc.
- Conversion to volume data: scanned (MRI/CT), numerical simulation (hydrogen data), or distance fields
- Rendering to screen (e.g. MIP – later slides)

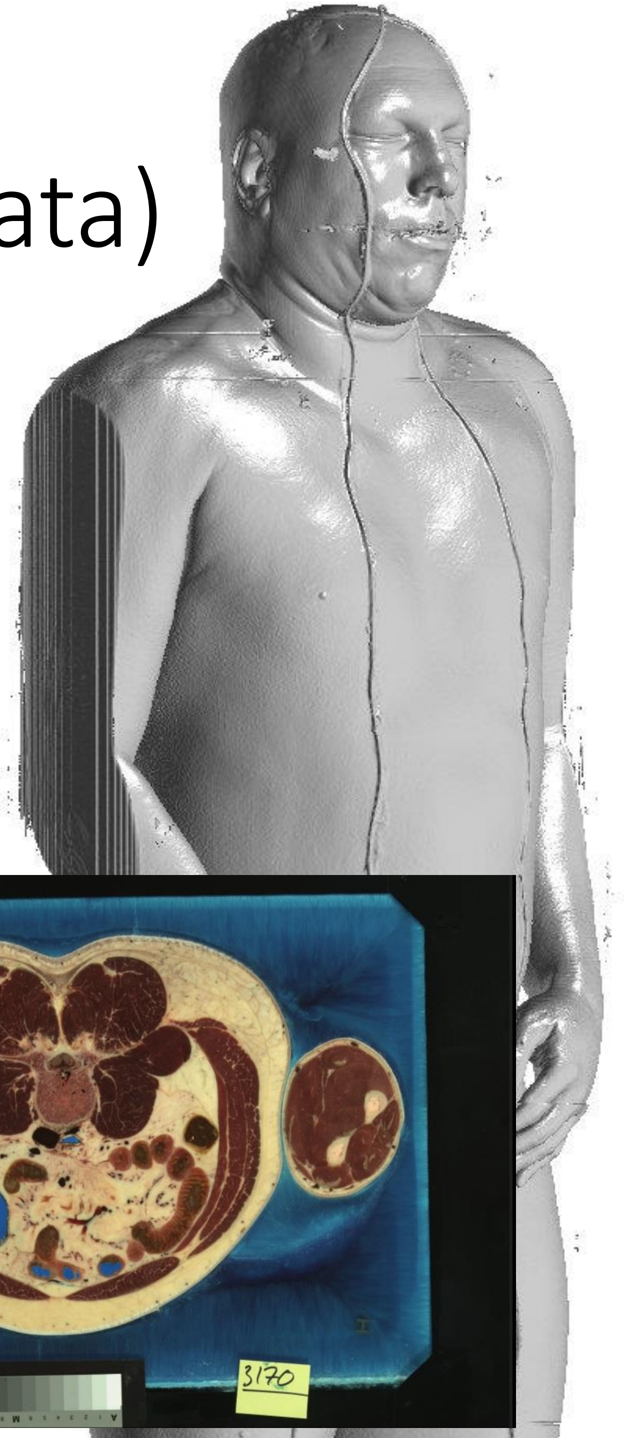


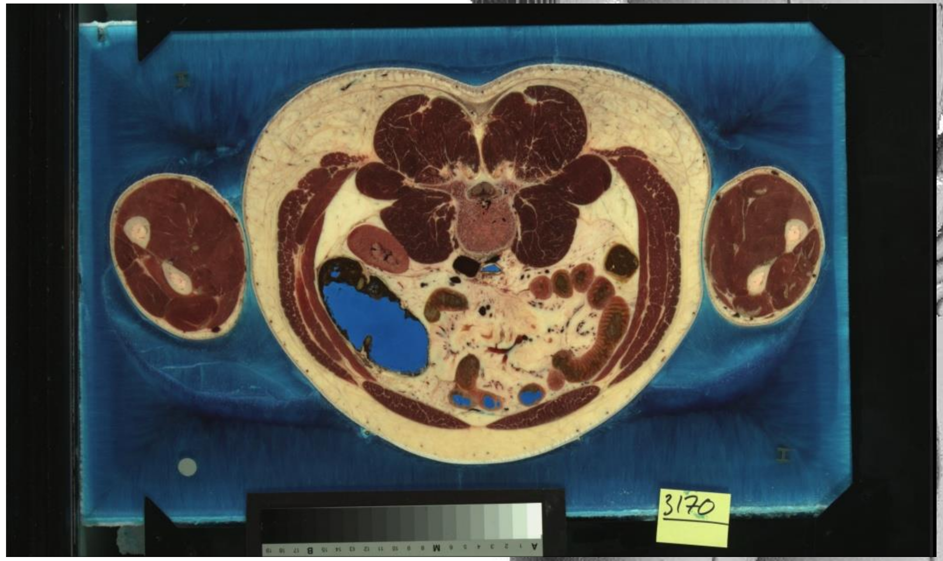
Hydrogen data


Distance around object

Visible Human Project (Volume Data)

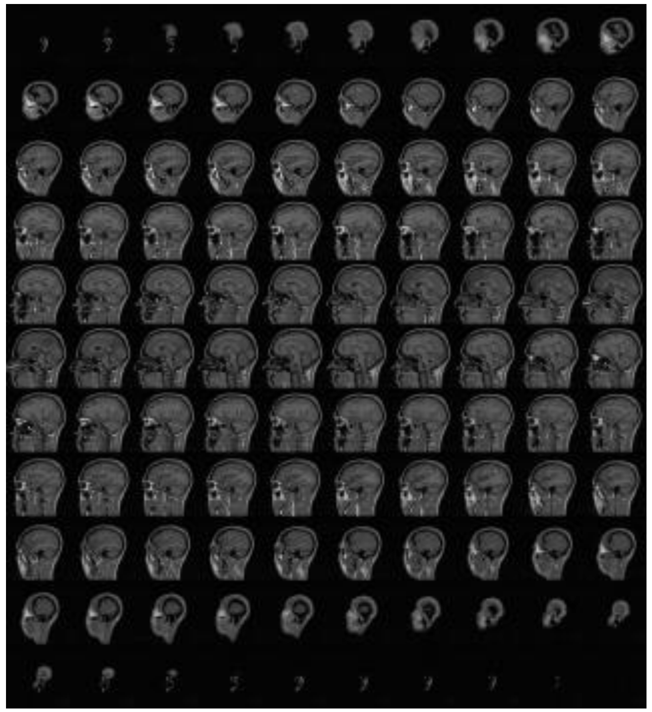
- CT scan 1800+ slices, each 512x512 voxels. Each voxel is 2 bytes (integer) ~ 1GB. MRI scan similar
- Body then photographed at 1800+ 1mm intervals (each photograph represents a slice through the body)





Volume Rendering Input

- The data consists of a number of images representing slices through the body
- Each pixel in each image has the measured quantity (X-ray absorption for CT scans)
- Stacking the images creates a 3D array of “voxels” = volume elements (pixels=picture elements)

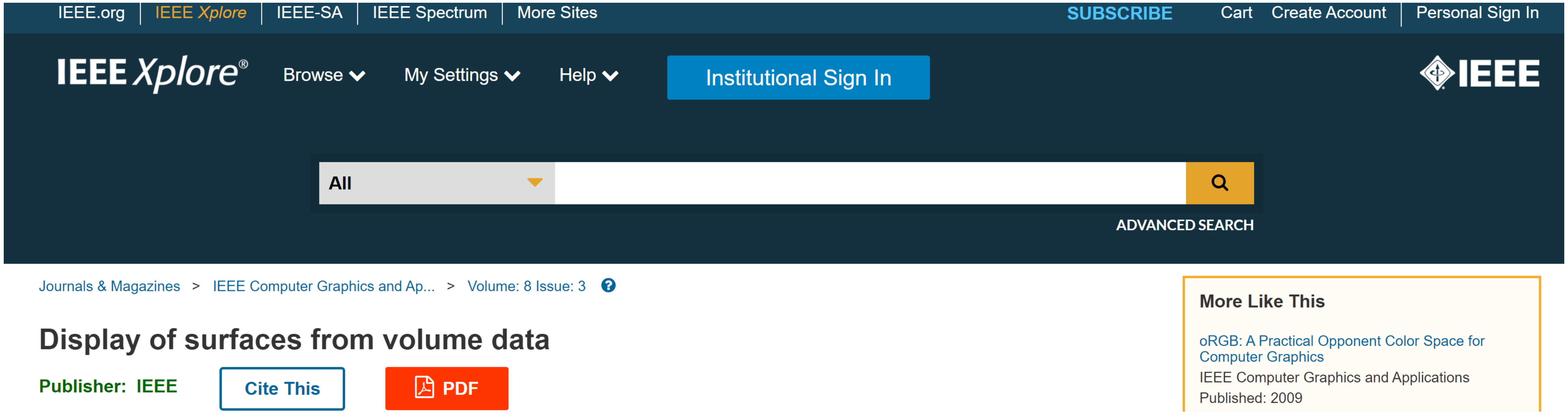


Volume Rendering (history)

- I wrote a chapter on Volume Ray Casting for my PhD thesis (Chapter published in a BCS conference). The earliest volume rendering papers are:
- M. Levoy. Display of surfaces from volume data. IEEE Computer Graphics and Applications, 8(3):29–37, May 1988.
- P. Sabella. A rendering algorithm for visualizing 3D scalar fields. In Proc. SIGGRAPH ’88 (Atlanta, Georgia, August 1-5, 1988), volume 22(4), pages 51–57. ACM SIGGRAPH, New York, August 1988.
- C. Upson and M. Keeler. V-Buffer : Visible volume rendering. In Proc. SIGGRAPH ’88 (Atlanta, Georgia, August 1-5, 1988), volume 22(4), pages 59–64. ACM SIGGRAPH, New York, August 1988.
- L. Carpenter R. A. Drebin and P. Hanrahan. Volume rendering. In Proc. SIGGRAPH ’88 (Atlanta, Georgia, August 1-5, 1988), volume 22(4), pages 65–74. ACM SIGGRAPH, New York, August 1988.

Levoy's Volume Rendering paper

- You can access Levoy's paper from the link.
- Other versions do not have the images from the original.
- To access click on Institutional Sign In



Journals & Magazines > IEEE Computer Graphics and Ap... > Volume: 8 Issue: 3

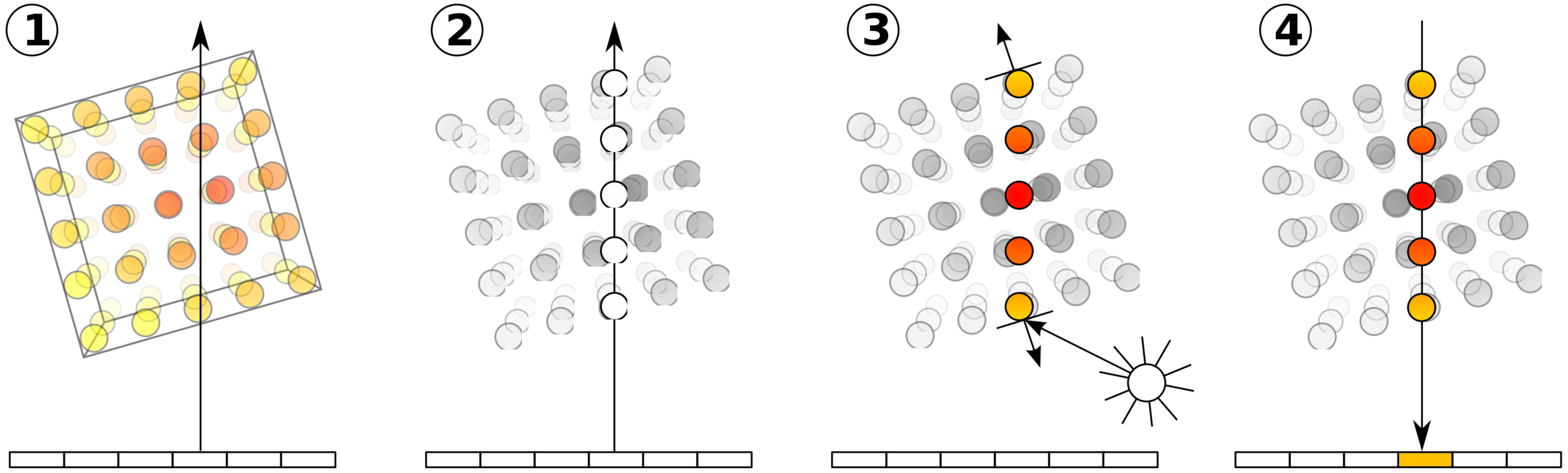
Display of surfaces from volume data

Publisher: IEEE

More Like This

oRGB: A Practical Opponent Color Space for Computer Graphics
IEEE Computer Graphics and Applications
Published: 2009

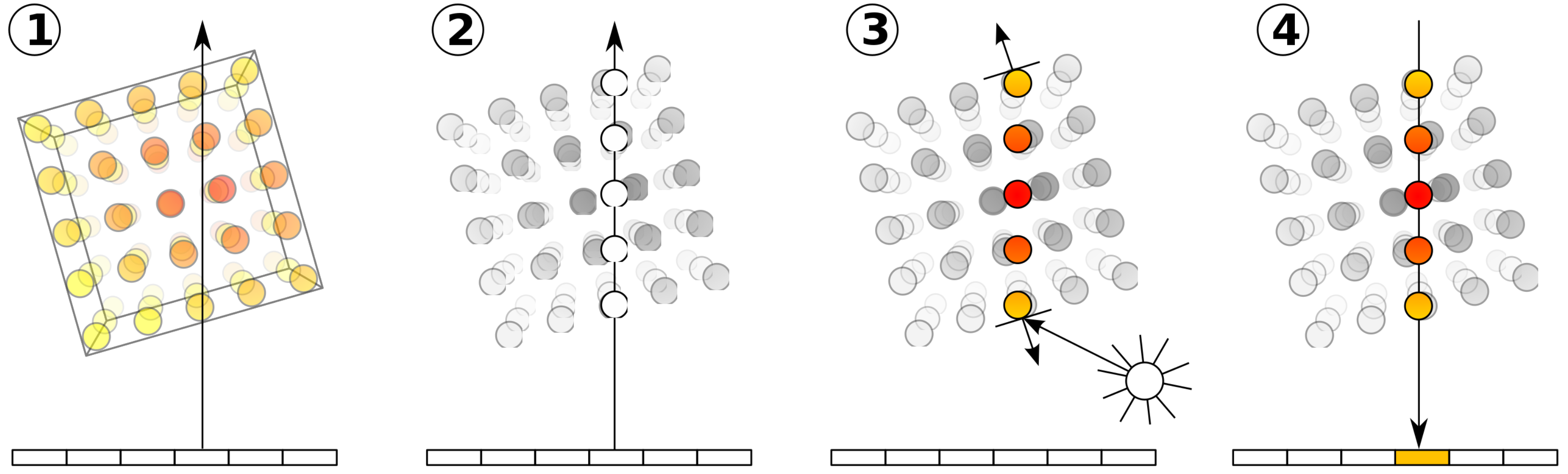
Volume Rendering (volume ray casting)



Volume_ray_casting.png: Thetawavederivative work: Florian Hofmann - Volume_ray_casting.png, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=14521474

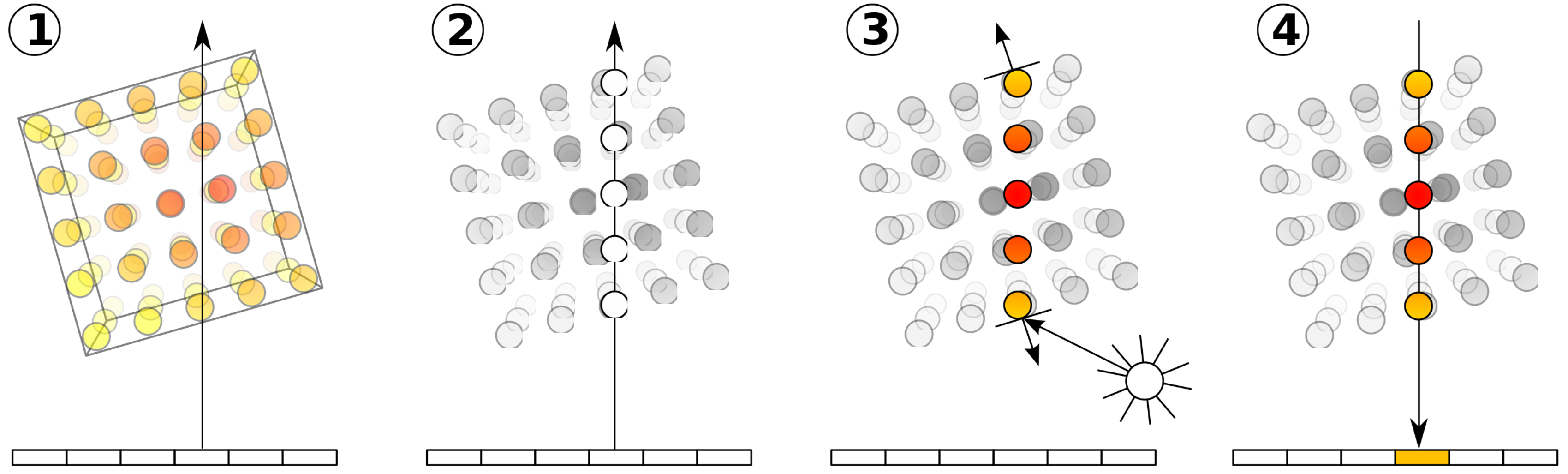
Four steps (adapted from Wikipedia)

1. **Ray casting**. For each pixel of the final image, a ray of sight is shot ("cast") through the volume. At this stage it is useful to consider the volume being enclosed within an axis-aligned bounding box (AABB) and use a Ray-AABB intersection test. *For the assignment we will not need to intersect with the volume because our cameras are fixed to top-down, side or front views and thus we can avoid this.*



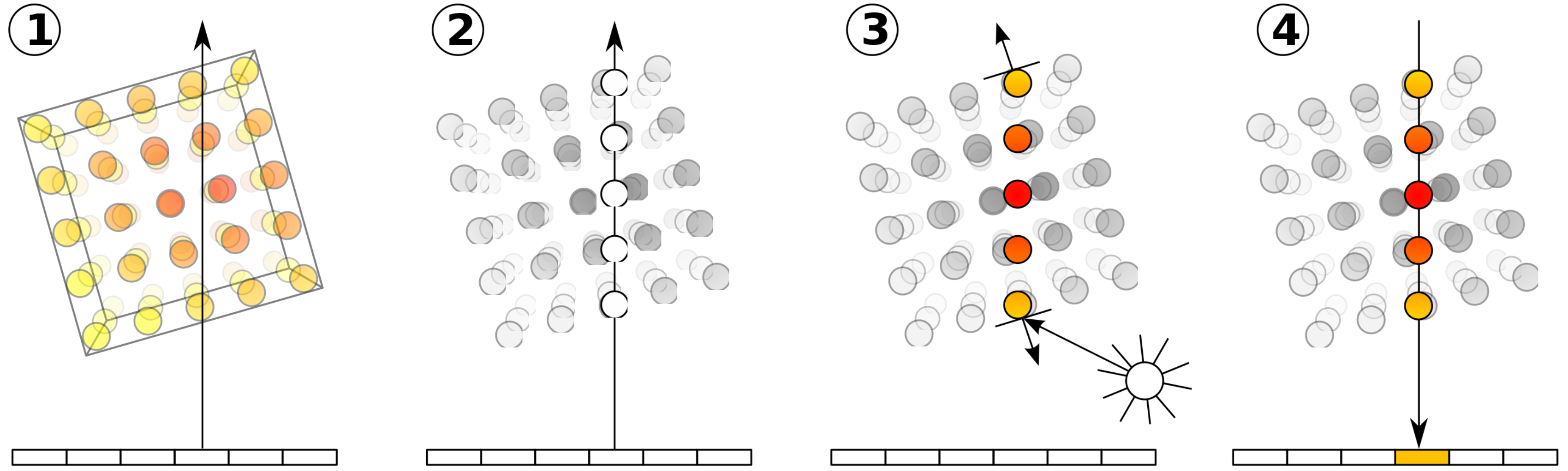
Four steps (adapted from Wikipedia)

2. **Sampling**. Along the part of the ray lying within the volume, equidistant samples are selected. *In the assignment, these are integer positions e.g., Cthead[z][y][x] where (x,y) are the pixel being traced in the top-down view and z is the current slice number being evaluated*. *We do not need to deal with the general case:* In general, the volume is not aligned with the ray of sight, and sampling points will usually be located in between voxels. Because of that, it is necessary to interpolate the values of the samples from its surrounding voxels (commonly using trilinear interpolation). *Interpolation is covered in the "Resizing images" lecture – you may choose this for Q3.*



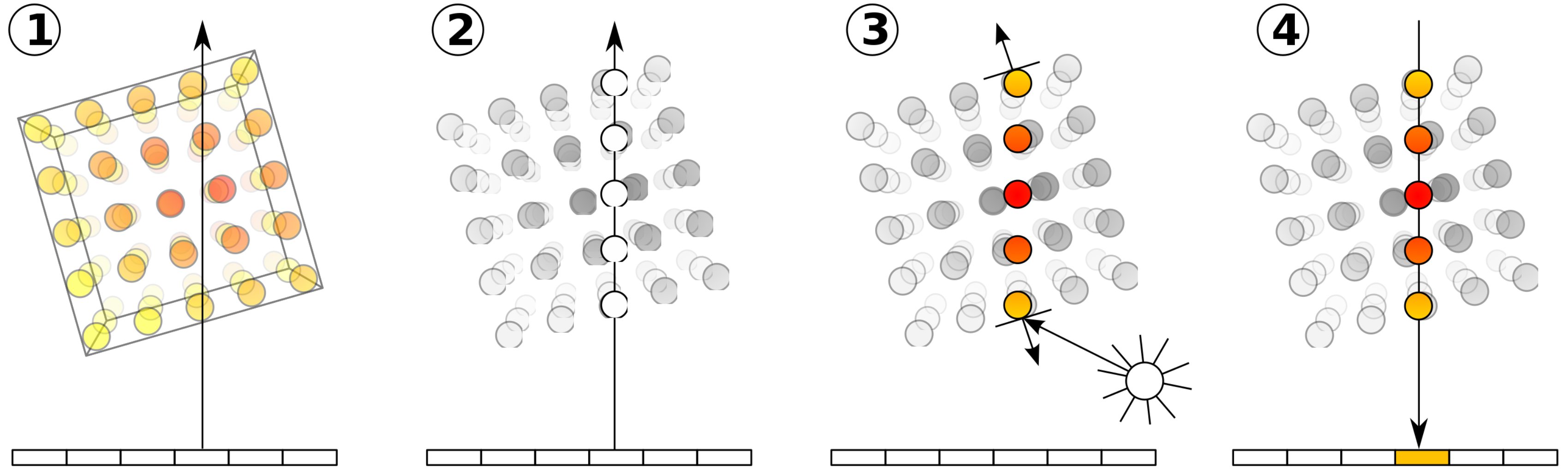
Four steps (adapted from Wikipedia)

3. **Shading**. For each sampling point, a transfer function retrieves an RGBA material colour (*we need this in the assignment*) and a gradient is computed (*you might choose this for Q3*). The gradient represents the orientation of local surfaces within the volume. The samples are then shaded (i.e. coloured and lit) according to their surface orientation and the location of the light source in the scene. *(Advice on how to do this is in the assignment sheet).*



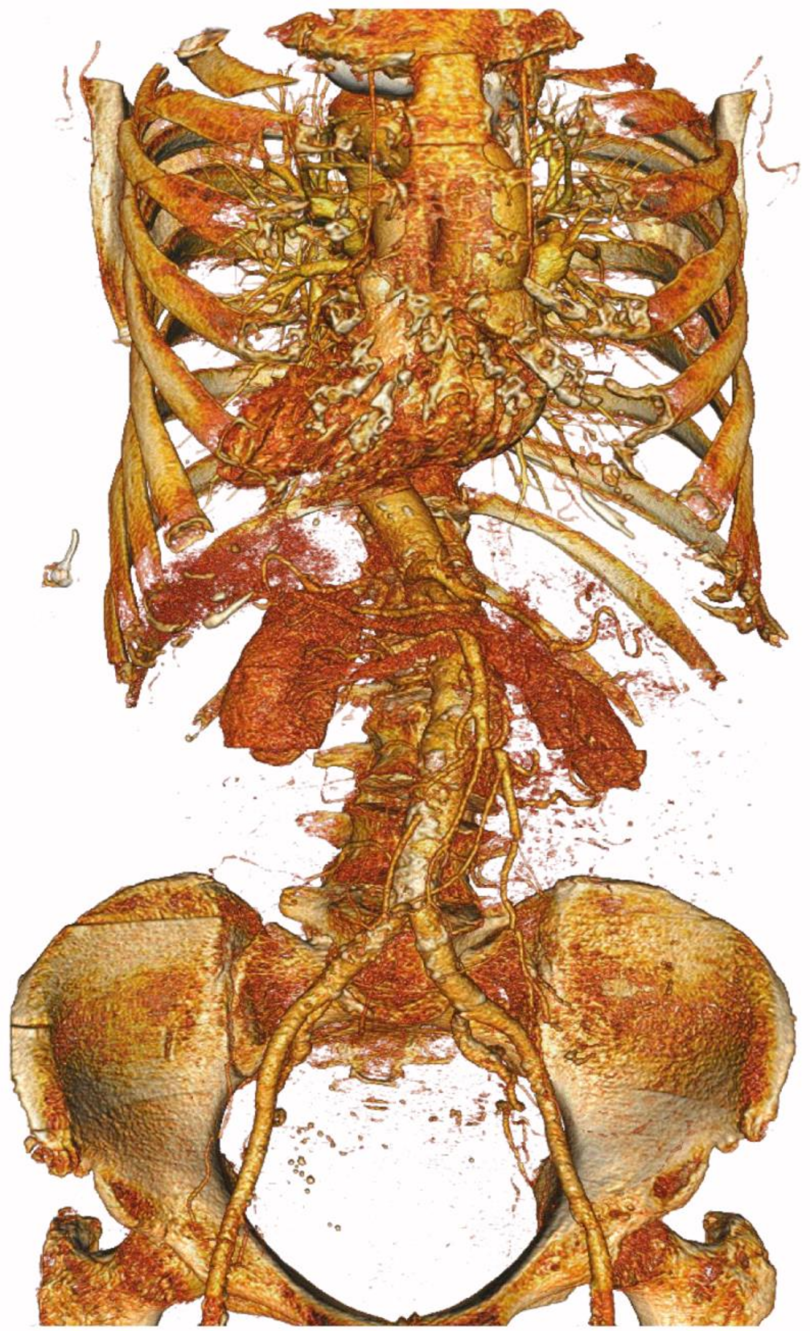
Four steps (adapted from Wikipedia)

4. **Compositing**. Colours for each sample are composited. Think of each sample as a piece of stained glass with lots of them in a row. It may work back-to-front, i.e. farthest sample first (easier to understand). Or, more efficiently, front to back.



Transfer function (TF)

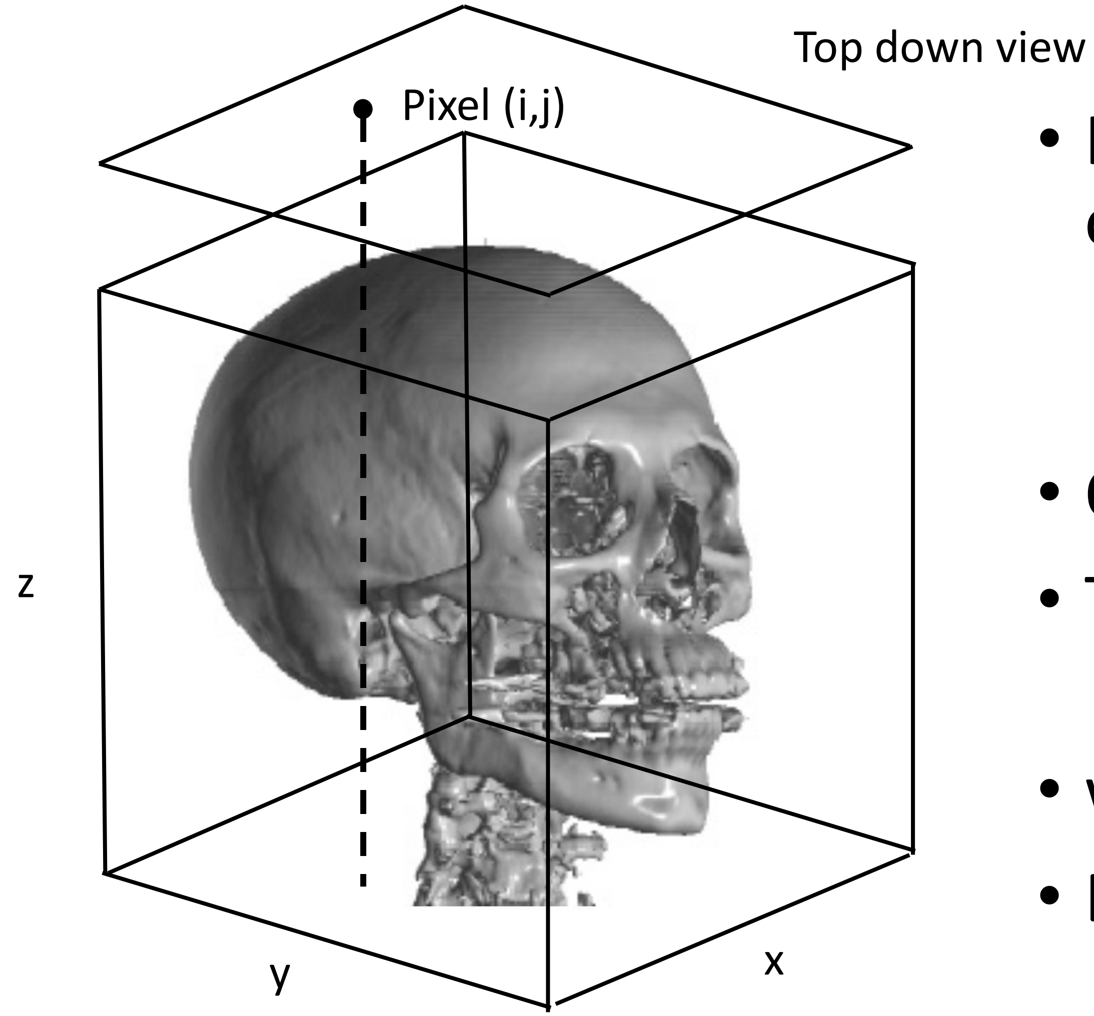
- The volume data contains an intensity value for each voxel.
- The values are e.g., X-ray absorption of the material.
- The transfer function maps voxel value to colour.
- We make a TF such that the colour matches the material (which had that voxel value).
- High quality images (like this one) require a good TF and good gradient shading.



Question: How do I start to code this?

- Let’s start in a way that we can develop the code in small steps.
- For each step we will get visual confirmation the algorithm is working so far (to make debugging easier than doing it all in one go).
- Step 1:
- Understand how to send rays through the volume.
- Step 2:
- While sending rays through the volume – test if there is a value >400 (bone) on our ray. If there is, make it white, otherwise, make it black.
- At this stage we should get a white head on a black background.

Casting rays through the volume



- Have the usual two outer loops to range over every pixel of the top down image

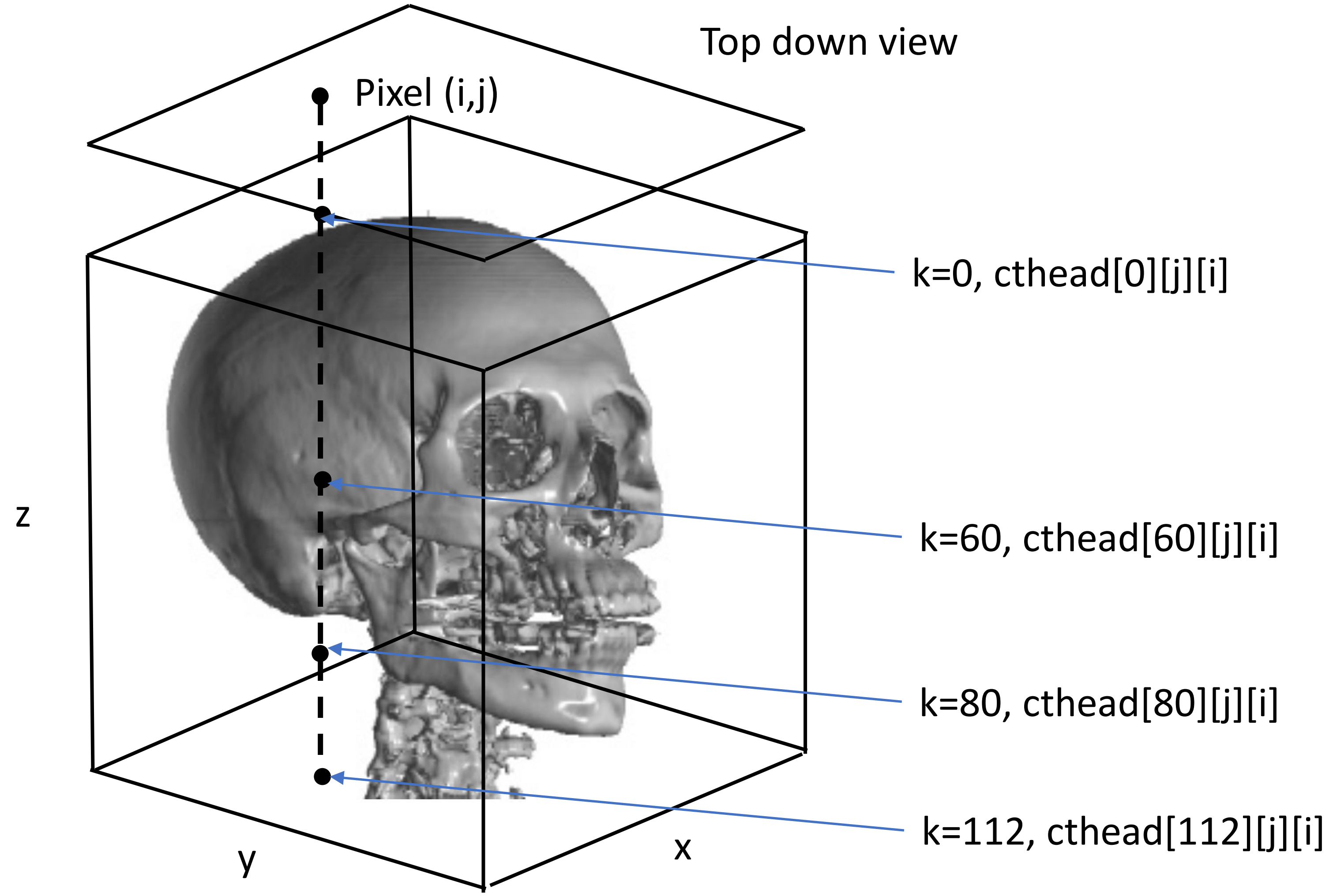
```
for j=row 0 to image height
    for i=column 0 to image width
```

- One example pixel is shown (i,j)
- Then an inner loop

```
for k=slice 0 to slice 112
```

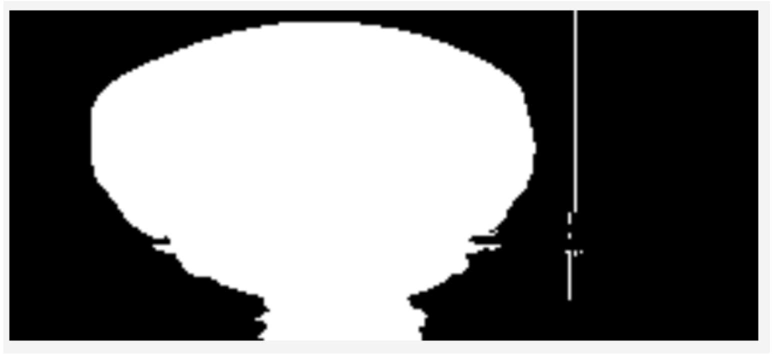
- will range over every voxel along the ray
- Do a computation on cthead[k][j][i]

Casting rays through the volume

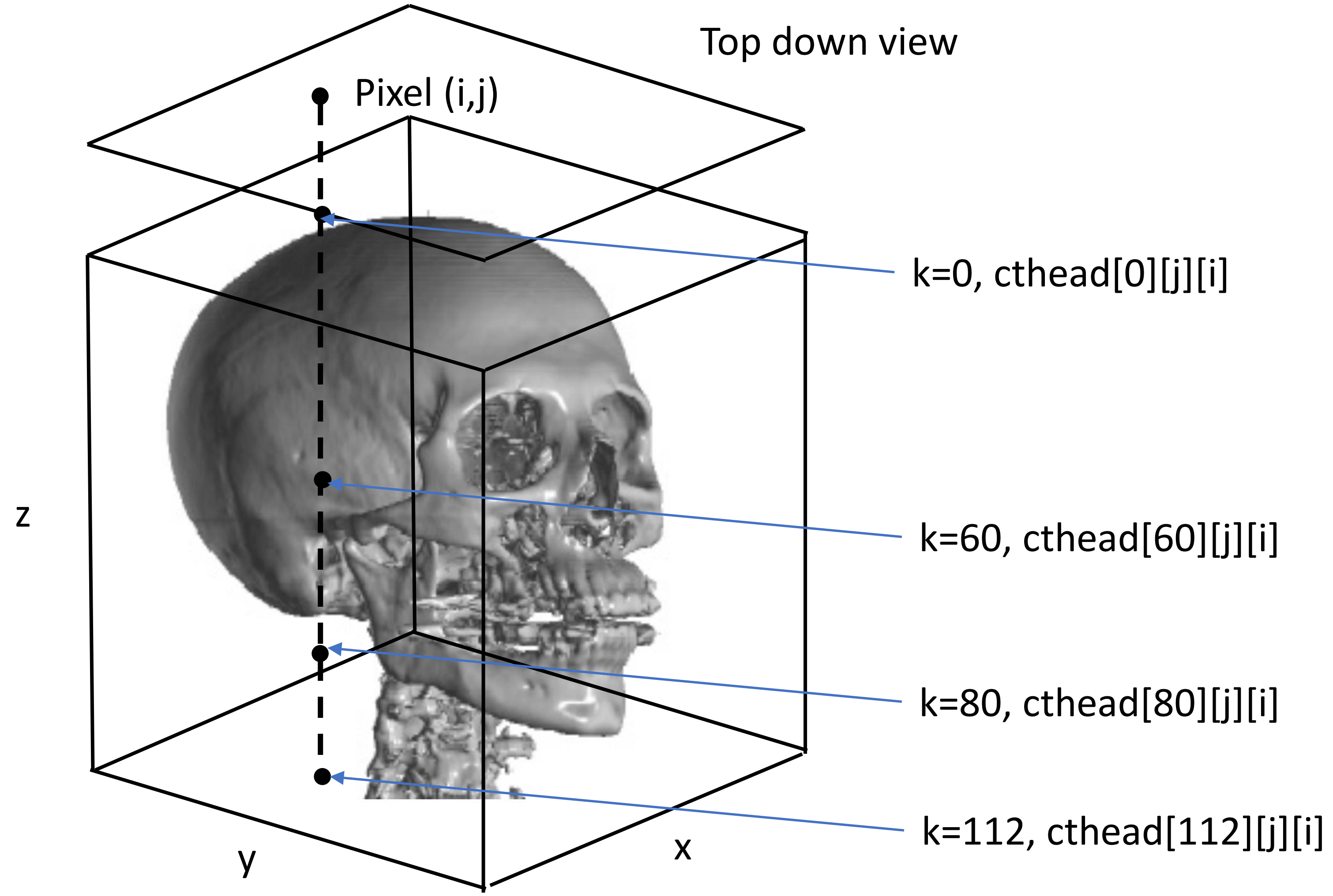


You can see the loop will visit each voxel along the ray.
Do a computation on cthead[k][j][i] to inform the colour at (i,j) on the image (top down view in this case).

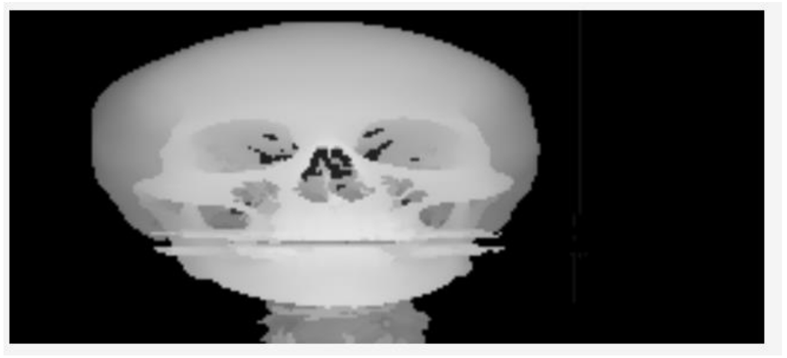
Example, set a value to 0 before the loop and set it to 1.0 if cthead[k][j][i]>400 at any point in the loop.



Depth based shading

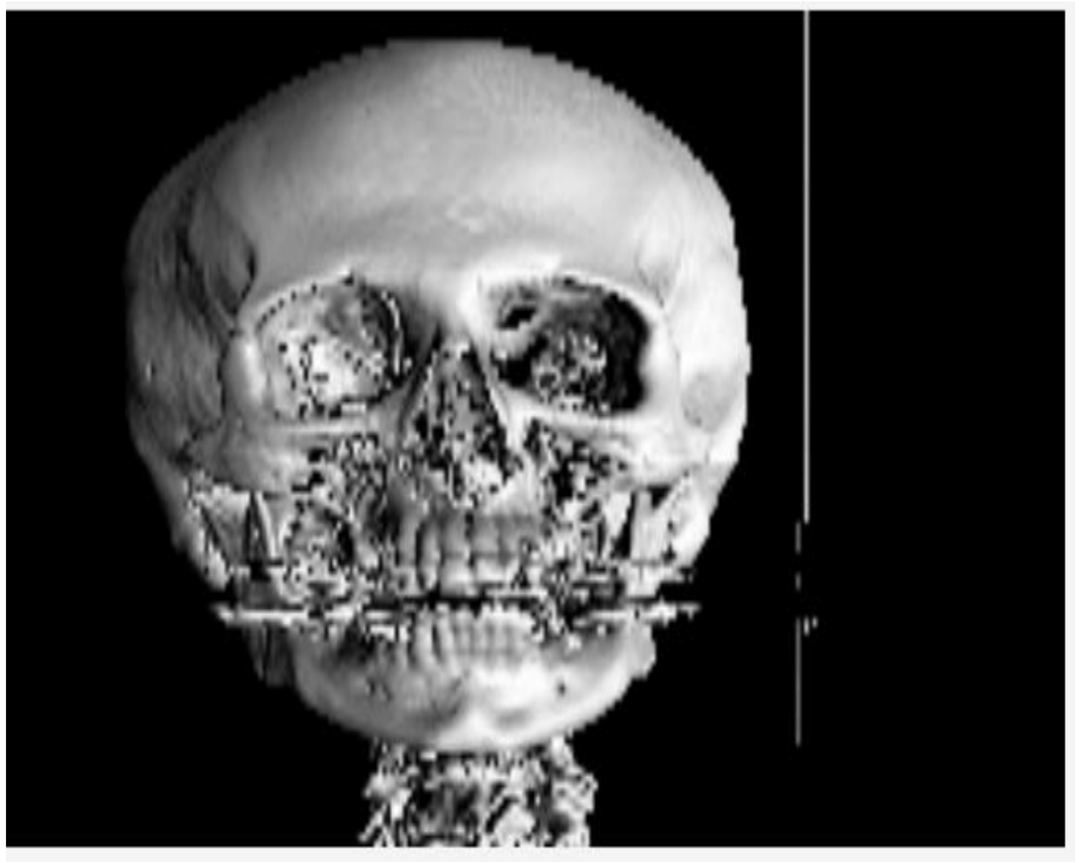


Depth based shading was developed early on in volume rendering.
The value of k for the top down view is 0 to 112 represents how far we are into the data.
Instead of setting the pixel to white, set it to k/112 or k/255 for other views. Here's the front view with pixels set to 1.0-k/255.
This is not advanced enough for Q3.

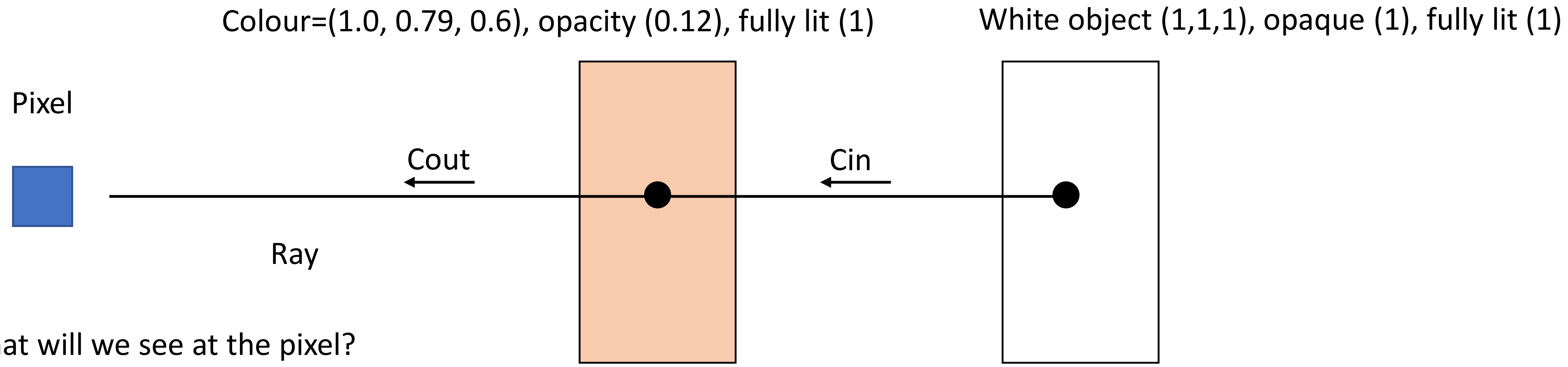


Aside: Gradient shading

- At this stage we could extend it using gradient shading.
- Calculate the gradient. Normalise it. Take the dot product with the (normalised) vector to the light source. My light is to the middle right.
- This is not perfect at this stage because the hit points are the first integer position after the voxel goes above 400.
- We can find more accurate positions using interpolation.
- The non-perfect way would gain marks in Q3. Accurate positions using interpolation (and corresponding gradients) would get full marks in Q3. It will look like the skull on the preceding slides.



Compositing


Colour=(1.0, 0.79, 0.6), opacity (0.12), fully lit (1)
White object (1,1,1), opaque (1), fully lit (1)
Pixel
Cout
Cin
Ray

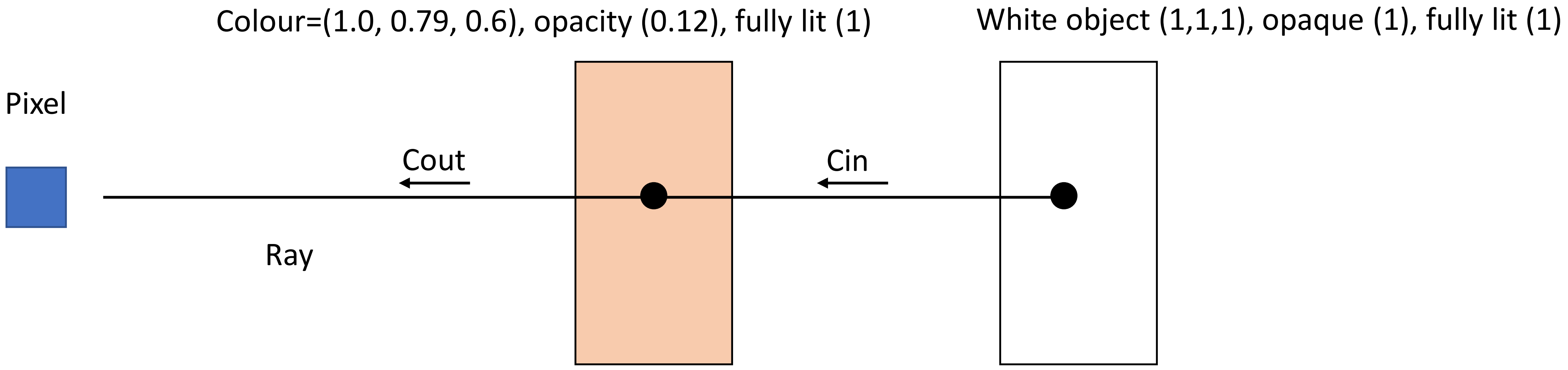
What will we see at the pixel?
The first object is nearly transparent (low opacity). We will mostly see white opaque object.
Try to work out the equation.
We will find a recurrence relation. The pixel colour will be the light contributed by the first thing seen on the ray plus all the light from the objects behind (and so on).
Assume we already know the colour of everything behind the first thing to be seen and this is C^{in}. We want to find the combined colour (C^{out}) of the closest object combined with C^{in}. The opacity of the first object is σ. The lighting factor is L. L could be the depth we calculated in depth based shading, the diffuse illumination from gradient shading, or start off using L=1.0.

$$C^{out} = \sigma L C + (1 - \sigma) C^{in}$$

Transfer function

- cthead[k][j][i] gives us a value (Hounsfield value)
- We need a colour mapping from HU to colour and opacity
- The cthead data set has values -1117 to 2248
- We need some code that sets the colour and opacity given a value in that range
- This code is up to you (e.g., you could use if..else statements in a function or a look up table (LUT)) – I prefer a LUT because it can be more efficient.
- The values I suggest are in the assignment sheet. You can change these to try to get different effects, but the ones given can be checked against the provided images and give you something to look at for debugging purposes.

Compositing (back to front = inefficient)


Colour=(1.0, 0.79, 0.6), opacity (0.12), fully lit (1)
White object (1,1,1), opaque (1), fully lit (1)
Pixel
Cout
Cin
Ray

$$C^{out} = \sigma L C + (1 - \sigma) C^{in}$$

Imagine there are many samples behind the white object.
They will have no impact on the colour of the pixel because the white object is fully opaque.
So any compositing of those values is a waste of computation.
Back to front is easier to understand, but inefficient.

Compositing (front to back = efficient)

We will traverse samples from the pixel (front) to the back of the data.
We will accumulate the transparency of all the samples so far. (Note: transparency=1-opacity: $\alpha = (1 - \sigma)$).
At the start (before the ray casting loop), we set the transparency α^{accum}=1 (there is nothing in the way of the first sample).
We will accumulate the colour of all the samples so far.
At the start (before the ray casting loop), we set the accumulated colour C^{accum}=(0,0,0). No colour.
In the loop we will add α^{accum} of the colour of the current sample σLC to the accumulated colour
Think about this:

For the first sample there is nothing in the way and we will see its full contribution.

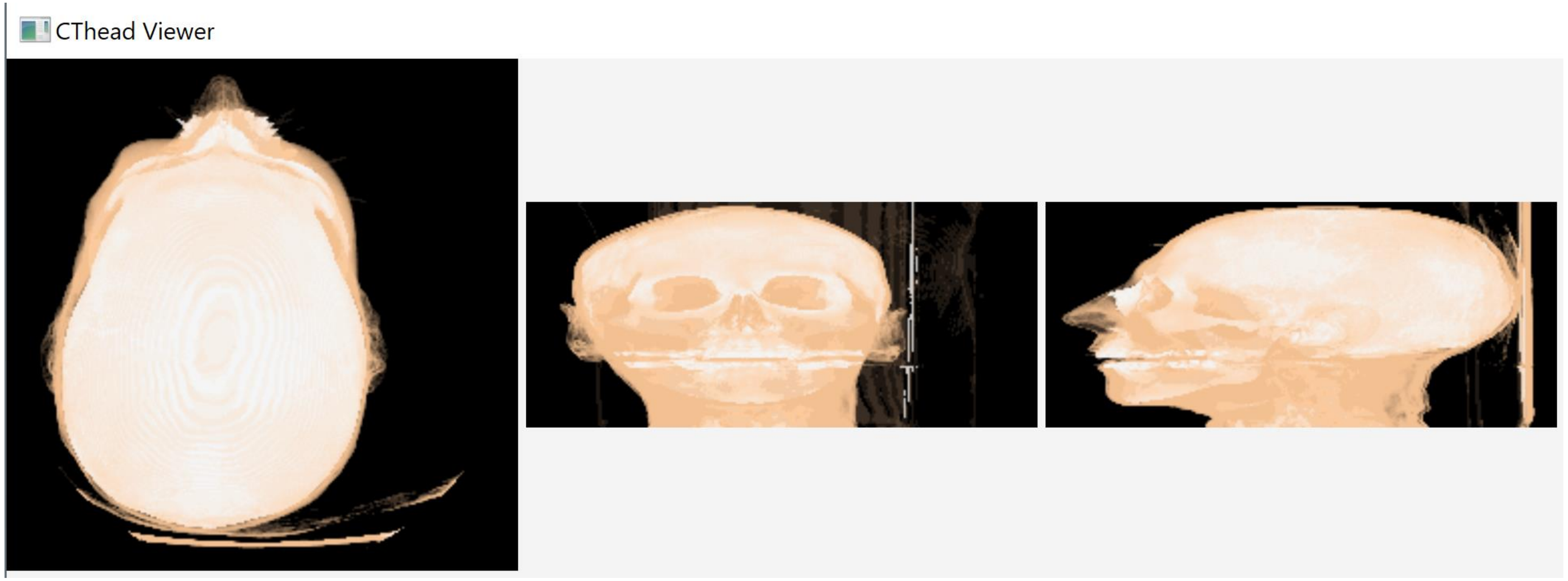
For the next sample we will see $(1 - \sigma)$ of its colour because the current sample has an opacity of σ.

So in the loop modify the accumulated transparency by how transparent the current sample is, $(1 - \sigma)$.
$\alpha^{accum} = \alpha^{accum} \times (1 - \sigma)$
We can now continue the loop with the next sample

Initialise: α^{accum}=1, C^{accum}=(0,0,0)
Loop over all the samples on the ray, doing:
$C^{accum} = C^{accum} + \alpha^{accum}\sigma LC,\ \alpha^{accum} = \alpha^{accum} \times (1 - \sigma)$

Remember a=a+b can be written in Java, C and C++ as a+=b

- Top down, front-to-back
- This is what the images look like for front to back rendering. The top down view (left) has been done using front-to-back and the inner k loop has gone from zero to CT_z_axis.



Debugging graphics code

- You could print variable values to make sure they are as expected.
- If you put a print inside the inner ray loop, it will be printed 7 million times (not helpful).
- Try, e.g.,

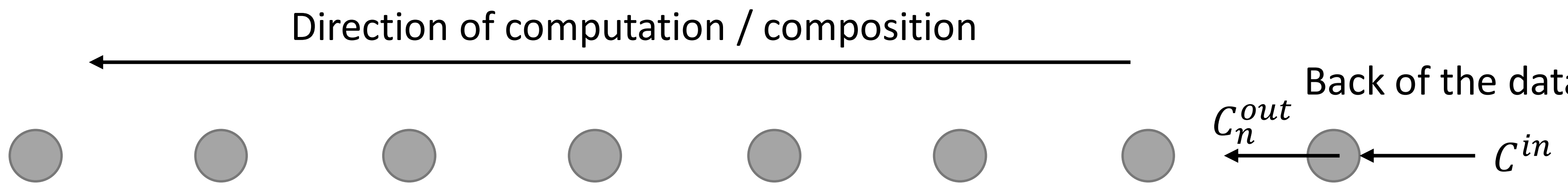
```
for (j=0; j<h; j++) {
   for (i=0; i<w; i++) {
      for (k=0; k<CT_z_axis; k++) {
         if (i==128 && j==128) {
            //print some values
```

Change pixel (128,128) to some other value of interest (i.e., this will not work on the shorter images).

Inside the inner k loop, using this approach, the values will be printed for every point on a single ray = CT_z_axis values (113)

These will be more manageable to look through and debug.

Back to front worked example



Back of the data set

n^{th} Sample colour

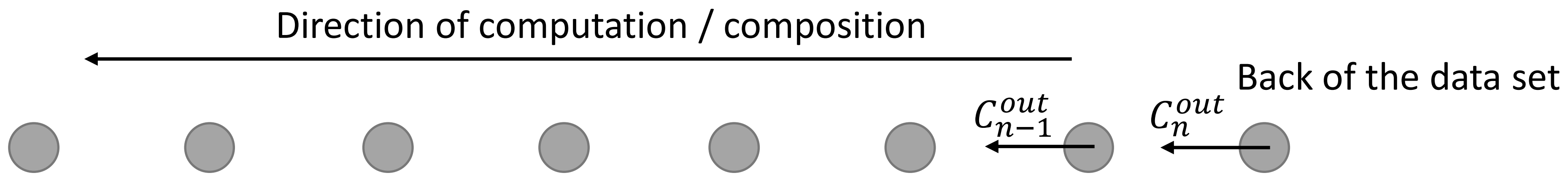
$$C_n = (r, g, b)$$

Sample opacity = σ_n

C^{in} into this sample is (0,0,0) because this is the “first” sample. I say “first” in quotes because it is the first one we encounter when looping from back to front – i.e. it is the last sample on the ray at the back, but the first we deal with in the back to front algorithm.

$$C_n^{out} = \sigma_n L C_n + (1 - \sigma_n) C^{in}$$

Back to front worked example



$n-1^{th}$ Sample colour

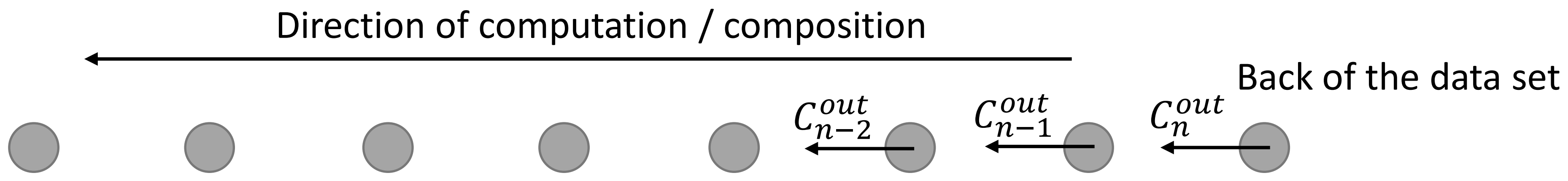
$$C_{n-1} = (r, g, b)$$

Sample opacity = σ_{n-1}

C^{in} into this sample the C_n^{out} from the previous sample.

$$C_{n-1}^{out} = \sigma_{n-1} L C_{n-1} + (1 - \sigma_{n-1}) C_n^{out}$$

Back to front worked example



$n-2^{th}$ Sample colour

$$C_{n-2} = (r, g, b)$$

Sample opacity = σ_{n-2}

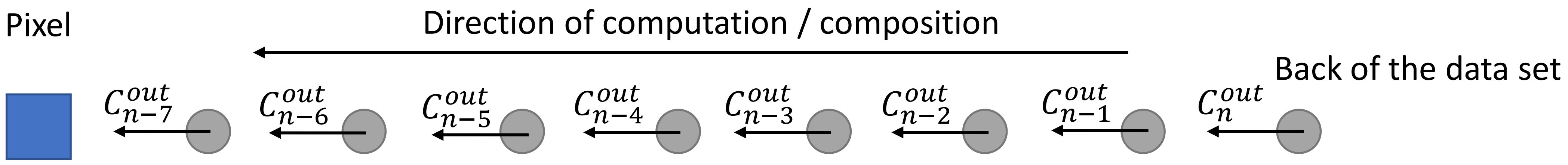
C^{in} into this sample the C_{n-1}^{out} from the previous sample.

$$C_{n-2}^{out} = \sigma_{n-2} L C_{n-2} + (1-\sigma_{n-2}) C_{n-1}^{out}$$

At each step we composite:

(1) the colour behind the current sample
(2) with the colour of the current sample
(3) in the ratio $\sigma_{n-2}: (1-\sigma_{n-2})$

Back to front worked example



$n-7^{th}$ Sample colour

$$C_{n-7} = (r, g, b)$$

Sample opacity = σ_{n-7}

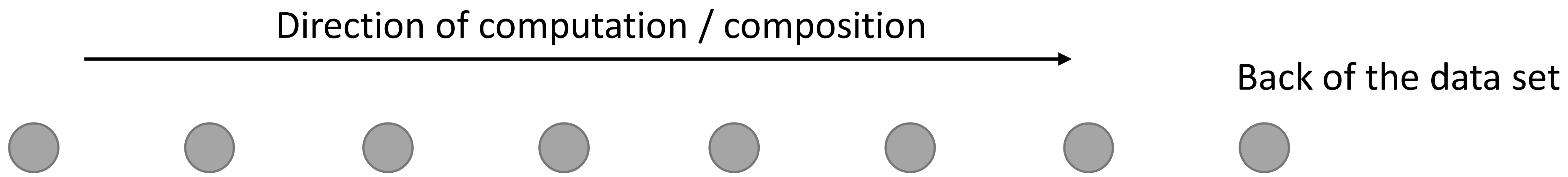
C^{in} into this sample the C_{n-6}^{out} from the previous sample.

$$C_{n-7}^{out} = \sigma_{n-7} L C_{n-7} + (1 - \sigma_{n-7}) C_{n-6}^{out}$$

This is the compositing equation applied in back to front manner

$$C^{out} = \sigma L C + (1 - \sigma) C^{in}$$

Front to back

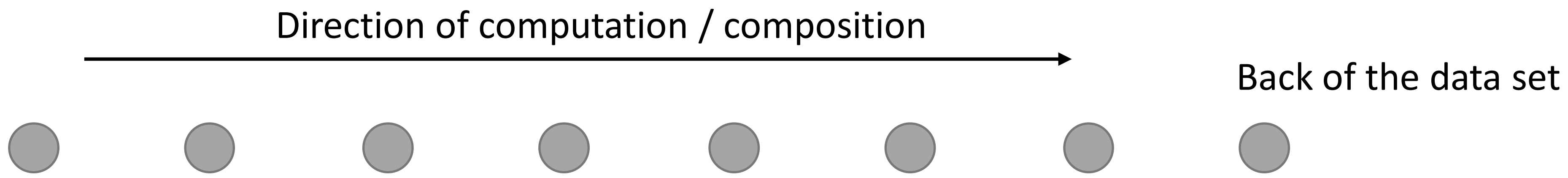


Initialise: α^{accum}=1, C^{accum}=(0,0,0)
L=1 in simple case (assume full lighting)

1^{st} Sample colour: $C_1 = (r, g, b)$
Sample opacity = σ_1

C^{accum}= $C^{accum} + \alpha^{accum}\sigma_1 L C_1$,
$\alpha^{accum} = \alpha^{accum} \times (1 - \sigma_1)$

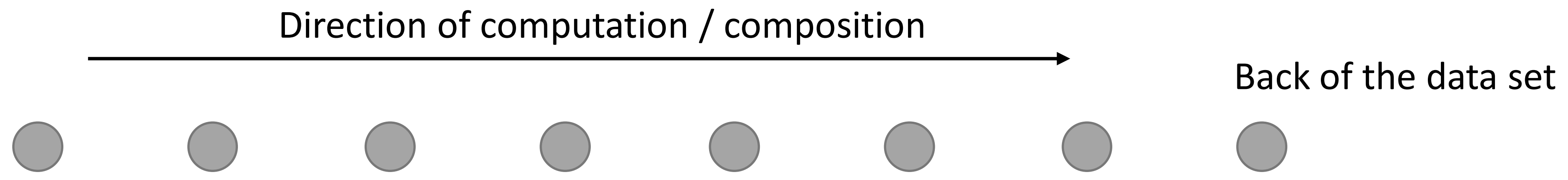
Front to back



2^{nd} Sample colour: $C_2 = (r, g, b)$
Sample opacity = σ_2

$C^{accum} = C^{accum} + \alpha^{accum}\sigma_2 L C_2,$
$\alpha^{accum} = \alpha^{accum} \times (1 - \sigma_2)$

Front to back



3^{rd} Sample colour: $C_3 = (r, g, b)$
Sample opacity = σ_3

$C^{accum} = C^{accum} + \alpha^{accum}\sigma_3 L C_3,$
$\alpha^{accum} = \alpha^{accum} \times (1 - \sigma_3)$

…and so on. The front to back approach uses the accumulated transparency to provide the equivalent result to back to front computation