# 2IMV20 Visualization: Report Assignment 1

Lois Nijland (0860184), Joost Pieterse (0848231)

**Ray casting**

**MIP and compositing**

***Implementation MIP***

The implementation of maximum intensity projection uses large parts of the implementation of the slicer, which was already given. One difference is that we added a third nested loop which goes from 0 up to the maximum ray length, which is computed to be the length of the diagonal of the volume. This third loop is introduced for the following:

pixelCoord[0] = uVec[0] \* (i - imageCenter) + vVec[0] \* (j - imageCenter) + volumeCenter[0] + (k - maxRayLength / 2) \* viewVec[0];

The part that is different from the slicer’s implementation is the last part: + (k - maxRayLength / 2) \* viewVec[0]. We use this to obtain the entire vector (all the points along the vector), whereas before, in the slicer, you only had one point. A similar implementation is done for pixelCoord[1] and pixelCoord[2].

The variable val is computed to be the maximum of all getVoxel(pixelCoord) or getInterpolatedVoxel(pixelCoord) (this will be described in Section Tri-linear interpolation). By doing this we get the maximum value for each point and we use these values for the final image.

The final difference between MIP and slicer is that some features were added for MIP to improve responsiveness. This is discussed in Section Responsiveness.

***Implementation compositing***

The implementation of compositing has only a few differences with MIP. We add a variable previousColor, which is initialized to be TFColor(0, 0, 0, 0). We do not use variable val anymore, instead we use variable alpha, which simply computes getVoxel(pixelCoord) or getInterpolatedVoxel(pixelCoord).

After this there is a new part, which is based on the “Front-to-Back” method for compositing. This method has the following formulas:

We implemented this “Front-to-Back” method as follows:

TFColor color = tFunc.getColor(alpha)

And we do for variable red the following (using the Cout formula above with our variables):

double red = previousColor.r + color.r \* alpha / max \* (1 - previousColor.a)

previousColor.r = red

For green and blue we do the same with previousColor.g, color.g and previousColor.b, color.b respectively. For newAlpha, we also do the same with previousColor.a, but here we do not use the “color.x” (using the αout formula above with our variables).

Now we set the “voxelColor.x” to be the computed “previousColor.x”. So voxelColor.r = previousColor.r, and similarly for voxelColor.g, voxelColor.b and voxelColor.a.

***Pros and cons***

A pro of MIP is that it displays the “inside” of the image. As an example, take the visualization of a skull. Using MIP, you will be able to clearly see the bone structure and teeth for example. So, because of this property the visualization of particular objects is very clear.

A con of MIP occurs when you want to visualize an object that has approximately the same intensity everywhere. If this is the case, then everything will appear in approximately the same color, and thus it will be difficult to distinguish between various parts of the object.

A pro of compositing is that you can very clearly see the outline of an object. So in the case discussed in the previous part, where the object has approximately the same intensity everywhere, you will still be able to see what is visualized very clearly. Another pro of compositing is that you can use different colors to make it clearer what object is visualized. For example, take the cross section of a lime, a lemon, and an orange. Using the color green for the lime, yellow for the lemon, and orange for the orange immediately shows you which object is visualized. Whereas, if you would not be able to use colors, these objects become almost impossible to distinguish.

A con of compositing is that some details of an image can get lost. Take again the example of the visualization of the skull. Certain parts, such as the teeth, cannot be seen in the final image when using compositing.

***Results***

Multiple comparisons are made to illustrate the strong and weak points of both MIP and compositing.

Figure XXX shows that the visualization of the pig is clearer when using compositing. The pig’s ears and snout can be distinguished much better from the rest of the pig when compositing is used. Also the flower pattern on the pig’s body can be observed better. This confirms the weak point of MIP; when the intensity is approximately the same everywhere, it becomes difficult to distinguish between the various part of the image. However, the compositing method is not better than the MIP method on all aspects. Namely, (a) reveals some lighter parts (coins) at the bottom of the pig. This cannot be seen in (b).

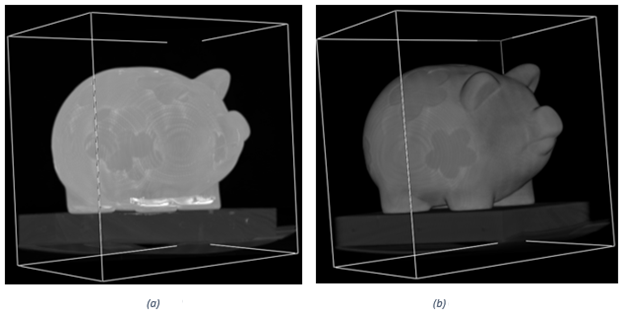


Figure Pig using (a) MIP and (b) Compositing

Figure XXX shows the difference between using MIP and compositing for the carp. These different techniques show different aspects of the carp. Figure XXX (a) visualizes the skeleton of the carp, whereas (b) and (c) visualize the outside of the carp. This illustrates the previously mentioned pro of MIP; it shows the “inside” of the carp. However, the outside of the fish, such as the head of the fish becomes less visible using MIP. In (b) and (c), the pro of compositing is shown; the outline of the carp is clearer.

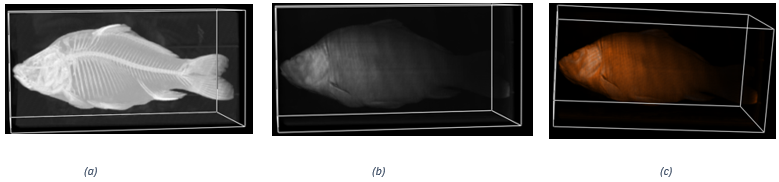


Figure Carp using (a) MIP, (b) Compositing, and (c) Compositing with colors

Figure XXX shows the top view of a backpack filled with different items. The difference between (a), (b) and (c) is not very large. In (a) the different items can be distinguished better than in (b) and (c). The items that are placed at the left, bottom and right sides are very similarly visualized. The items that are in the middle of the image become more blurry when using compositing instead of MIP. However, this does not happen when we use another angle to view the backpack, as can be seen in Figure XXX. This could indicate that some part of the backpack is blocking the view to the middle objects in Figure XXX (b).

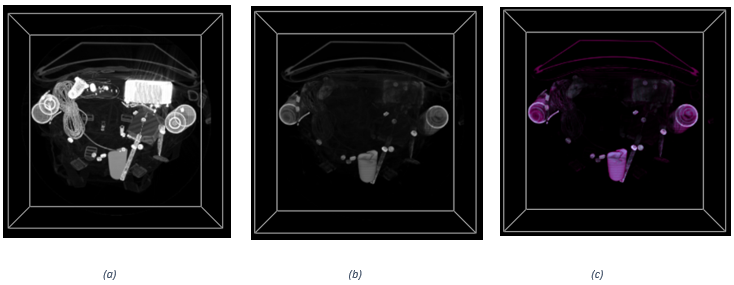


Figure Top view backpack using (a) MIP, (b) Compositing and (c) Compositing with colors

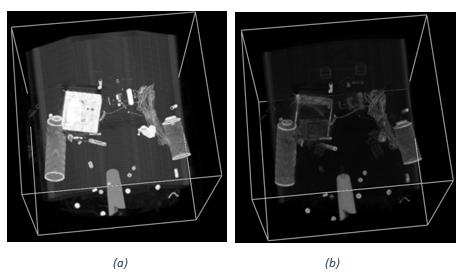


Figure Different view backpack using (a) MIP and (b) Compositing

Figure XXX visualizes a tooth. In (a) the tooth’s shape is clearly illustrated. The tooth appears to be in some kind of box; in (a) we see a grey box around the tooth. Remarkably (b) does not show any hint of what object is hidden in the grey box. This strongly illustrates the weak point of compositing, namely that some details of the image can get lost. In this particular case, the “detail” that gets lost is actually the most important part of the image.

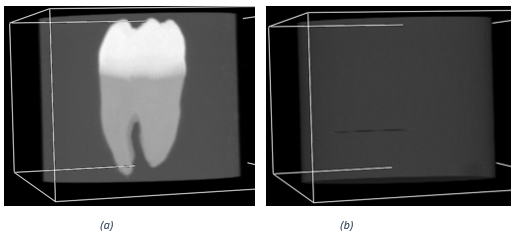


Figure Tooth using (a) MIP and (b) Compositing

Figure XXX visualizes a tomato. These visualizations show different aspects of the tomato. In (a) a cross section of the tomato can be seen. Figure XXX (b) and (c) display the outside of the tomato. The colored tomato illustrates the strong point of compositing mentioned above. Namely, the use of coloring can make it much clearer what is seen. For example, when you see (a) you will probably not immediately see that it is a cross section of a tomato. However, if you see (c) it is much more obvious that a tomato is shown.

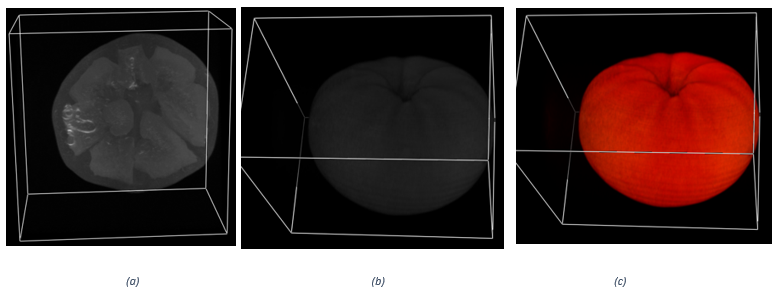


Figure Tomato using (a) MIP, (b) Compositing and (c) Compositing with colors

**Tri-linear interpolation**

***Implementation***

For the implementation of tri-linear interpolation we computed x0, x1, x2, x3, x4, x5, x6, x7 as in the picture given in the slides, which can also be seen in Figure XXX.

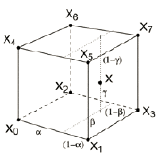


Figure Tri-linear interpolation model with points

To illustrate how exactly we computed these points, we show how we computed point x5:

int x5 = getVoxel(new double[]{pixelCoord[0] + 1, pixelCoord[1], pixelCoord[2] + 1})

The other points are computed similarly.

Then we compute alpha, beta and gamma. We do this as follows:

double alpha = pixelCoord[0] - Math.floor(pixelCoord[0])

This sets alpha to be the value after the decimal point of pixelCoord[0].For beta and gamma we use the same computation but instead of pixelCoord[0] we use pixelCoord[1] and pixelCoord[2] respectively.

Finally, we implemented the formula as given in the slides, namely:

Here we used the variables alpha, beta, gamma, x0 , x1, x2, x3, x4, x5, x6, x7, which we computed before.

***Results***

In Figure XXX two images are shown, where in (a) we do not use tri-linear interpolation and in (b) we do use it. As can be seen, there is no observable difference between these two images. This occurred for all the different images that were tested for this.

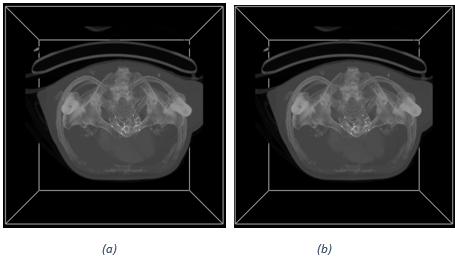


Figure Top view of stent, not using tri-linear interpolation in (a), and using tri-linear interpolation in (b)

**Responsiveness**

The raycaster becomes quite slow when using the application. To increase responsiveness during user interaction we use the provided variable interactiveMode. This indicates whether there is a lot of user interaction. We check if this is the case, so if interactiveMode = true. If so, we will not use the tri-linear interpolation method and we increase the variable step to n, whereas before this was 1. By increasing it to n we lower the resolution by n2, so if n=2, groups of 4 pixels have the same color. This significantly increases responsiveness. However, since we now only look at one n2th of the values, the resulting image will have a lower resolution. So, we had to make a tradeoff between responsiveness and the quality of the image. We did this by investigating different possibilities.

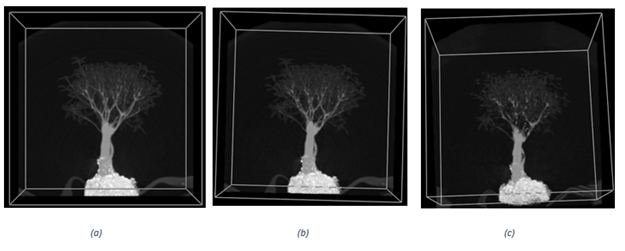


Figure In (a) step is 1, in (b) step is 4, in (c) step is 8

As can be seen in Figure XXX , the quality of images (a) and (b) does not differ very much, whereas image (c) differs significantly from image (a), particularly visible in the smaller branches of the tree at the top.

When interacting with these different step values, we found that the interaction of (a) is quite slow, the interaction of (b) is acceptable/okay and the interaction of (c) is somewhat faster than (b).

Therefore, we found that setting (b) gave the best resolution and interaction combination. So, we chose n to be equal to 4.

**2-D Transfer functions**

**Gradient-based opacity weighting**

***Implementation***

We implemented the gradient-based opacity weighting using Levoy’s formula. Figure XXX and XXX show the two parts of this formula.

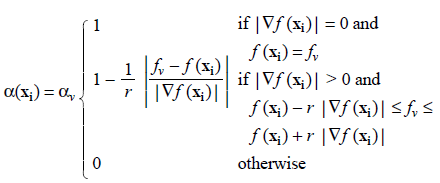
 

Figure Levoy's formula part 1 Figure Levoy's formula part 2

In our program we define the variables from the formula as follows:

|  |  |  |
| --- | --- | --- |
| Variable name in formula | What the variable represents | How it is used in our program |
| αv | Opacity in the widget | double alpha = tfEditor2D.triangleWidget.color.a; |
| r | Radius in the widget | double r = tfEditor2D.triangleWidget.radius; |
| fv | Intensity in the widget | double baseIntensity = tfEditor2D.triangleWidget.baseIntensity / max; |
| f(xi) | Result of getVoxel (or getInterpolatedVoxel) | intensity = getVoxel(pixelCoord) |
| |del f(xi)| | Length of the gradient | float gradientLength = gradient.mag / (float) max; |
| α(xi) | Final value of α for voxel xi | if (gradientLength == 0 && intensity == baseIntensity) { resultAlpha = alpha; }  else if (gradientLength > 0 && intensity - r \* gradientLength <= baseIntensity && intensity + r \* gradientLength >= baseIntensity) { resultAlpha = alpha \* (1 - (Math.abs(baseIntensity - intensity)) / (gradientLength \* r));}    else {resultAlpha = 0;} |
| 1-αtot(xi) | Product of all (1-αn(xi­)) | alphaProduct \*= 1 - resultAlpha; |
| αtot(xi) | Final value for α | voxelColor.a = 1 - alphaProduct; |

***Results***

Below multiple images are shown to illustrate the different capabilities of the 2D transfer function. Also comparisons are made between the 2D transfer function, MIP and compositing.

Figure XXX shows different images of the pig. In (a) we can see the coins at the bottom of the pig best. In (b) we see that the pig is placed on a block. Image (c) does not show the block or coins anymore, but shows the flower pattern on the pig. If we compare this with Figure XXX (mip and composite pig) we see that especially the coins inside the pig become much more visible when using the 2D transfer function. The pig’s outline in Figure XXX (below) (a) and (b) is blurrier than the pig’s outline in Figure XXX (mip and composite pig) (a) and (b). Figure XXX (below) (c) has a sharper outline. However, Figure XXX (composite pig) (b) still gives a slightly better view of the exterior of the pig.

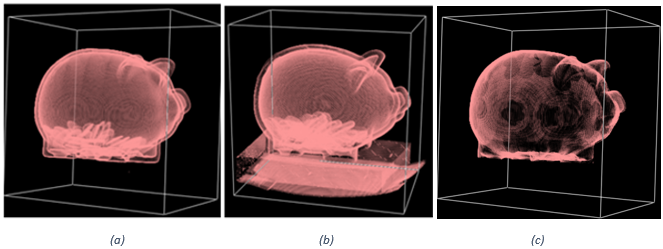


Figure Pig using different settings of 2D transfer function

In Figure XXX the carp is visualized using the 2D transfer function. In (a) the skeleton of the carp can be observed. This resembles Figure XXX (MIP) (a) strongly, where the MIP method is used. Figure XXX (b) is comparable to Figure XXX (compositing) (c), since they both visualize the outside of the carp. When comparing these two images, we see that in Figure XXX (compositing) (c) the head of the carp is visualized clearer, whereas in Figure XXX (b) the tail is visualized clearer. So, if we compare the 2D transfer function with both MIP and compositing for the image of the carp, we see that the 2D transfer function has more capabilities than MIP or compositing separately. Namely, because the 2D transfer function captures the skeleton of the carp as well as the outside of the carp. However, if we are only interested in the outside of the carp, then compositing might be the better alternative.

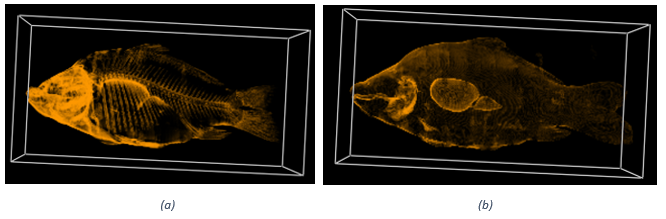


Figure Carp using different settings of 2D transfer function

Figure XXX shows the visualization of an orange. In (a) we can easily distinguish the different pieces of the orange. In (b) this is somewhat more difficult, but here we can observe the peel of the orange and the seeds that are inside. As in the previous examples, the 2D transfer function shows different aspects of the image. When comparing this with the MIP and compositing image of the orange (which were already given in the assignment, and therefore not included in our report) we can conclude that we gain information by using the 2D transfer function; the seeds are not visible in the other images.

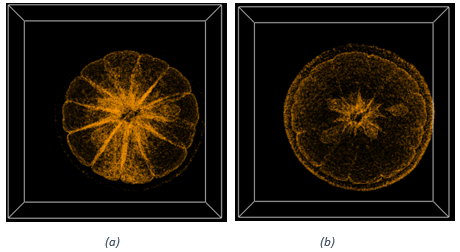


Figure Orange using different settings of 2D transfer function

In Figure XXX two images of the tomato are displayed. In (a) we can clearly see the inside, such as the seeds, of the tomato. If we compare this to MIP and compositing, as seen in Figure XXX (MIP, compositing), then we observe that these seeds etc. cannot be seen when using MIP or compositing. Regarding this, Figure XXX (a) more clearly shows that a tomato is visualized, so in this aspect the 2D transfer function gives the viewer more information. If we look at Figure XXX (b), then we can see the outside of the tomato. When comparing this with Figure XXX (MIP, compositing) (c), we conclude that Figure XXX (MIP, compositing) (c) visualizes this more clearly.

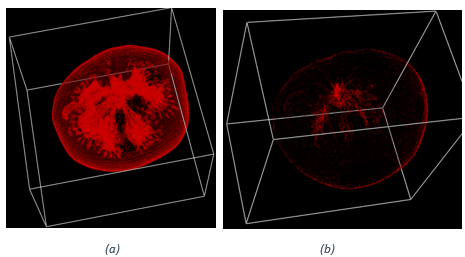


Figure Tomato using different settings of 2D transfer function

**Extended triangle widget**

***Implementation***

***Results***

**Illumination model**

***Implementation***

Our implementation of the shading function is based on the simplified Phong model. This uses a halfway vector:

It uses the following formula:

We initialize the following variables:

double kAmbient = 0.1

double kDiff = 0.7

double kSpec = 0.2

int power = 10

We implemented the formula as follows:

|  |  |
| --- | --- |
| Part of the formula | How it is used in our program |
| N | double[] normal = {gradient.x / gradient.mag, gradient.y / gradient.mag, gradient.z / gradient.mag}; |
| |L+V| | double halfwayLength = VectorMath.length(halfway); |
| H | double[] halfway = {2 \* viewVec[0], 2 \* viewVec[1], 2 \* viewVec[2]};  halfway[0] /= halfwayLength;  halfway[1] /= halfwayLength;  halfway[2] /= halfwayLength; |
| kdiff (L\*N) | double diffuseMultiplier = kDiff \* (VectorMath.dotproduct(viewVec, normal)); |
| Ia + Idkdiff (L\*N) | newColor.r = tfEditor2D.triangleWidget.color.r \* diffuseMultiplier + kAmbient \* tfEditor2D.triangleWidget.color.r;  newColor.g = tfEditor2D.triangleWidget.color.g \* diffuseMultiplier + kAmbient \* tfEditor2D.triangleWidget.color.g;  newColor.b = tfEditor2D.triangleWidget.color.b \* diffuseMultiplier + kAmbient \* tfEditor2D.triangleWidget.color.b; |
| kspec(N\*H)α | double spec = kSpec \* (Math.pow(VectorMath.dotproduct(normal, halfway), power)); |
| I | newColor.r += spec;  newColor.g += spec;  newColor.b += spec;  previousColor.r = newColor.r \* resultAlpha + previousColor.r \* (1 - resultAlpha);  previousColor.g = newColor.g \* resultAlpha + previousColor.g \* (1 - resultAlpha);  previousColor.b = newColor.b \* resultAlpha + previousColor.b \* (1 - resultAlpha); |

***Results***

**Comparison of techniques**

(Ik doe dit nu overal tussendoor al, dus dan hoeft dit stukje niet meer denk ik. We kunnen hier eventueel wel nog een opsomming geven van strong and weak points)

Compare the results obtained from various data sets of the different approaches

The comparisons should clearly demonstrate the strengths and weaknesses of each of the techniques.

For “results”: the techniques should be applied to several data sets, interesting details in the data should be reported by showing a good set of transfer functions. The exploration process should involve extensive experimentation with the parameters of the various approaches.

**Appendix**

In order to be able to recreate the pictures that we used in this report, we give these pictures below with their corresponding transfer functions.

