From closed-loop video to 3D Model via silhouettes

Bachelor Thesis

Bachelor Computer Science

Hans Wortel S0607940,
Thomas Dorsman S0601977
1 August 2009

Supervisor: dr. Nies Huijsmans

Universiteit Leiden

Introduction

For our bachelor thesis we decided to work on a project which turns a movie into a 3d object or volumetric scene. Quick reconstruction of volumetric scenes given a number of views is important for applications such as interactive visualization of environments and objects by a recording device[1]. It may provide the user with better human-readable data. For instance a model of a house with the same amount of detail as a set of images of a house, may be easier to study. Currently there is a large variety of techniques to create these volumetric scenes. Coherent to the technique used is the given parameters and how well their values are known. The best preparation, resulting in as much details as possible, would be one with as many known/constant parameters as possible. Some of these parameters are: Camera angle, movement of the camera relative to the object, background color and background scene. When these parameters differ and are not known it is often more difficult to render the volumetric scene and often less detail will be achieved. For instance: a recording from a F16 fighter of a target building does not have as many known parameters as a recording from a object on a LP player with a static camera. In the last case it is useful to use a technique called voxel sculpting or voxel carving, while the F16 recording would need a totally different approach like point recognition.

Voxel sculpting

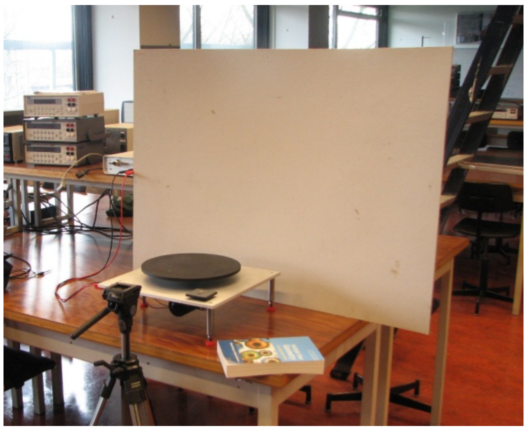


Figure 1. Our preparation.

For our project we choose the most detailed approach (as many known parameters as possible) using a fixed rotation turntable and a camera with a static viewpoint. We liked the idea that we would get realistic objects as result which we could use in programs like blender. For this reason we decided to use voxel sculpting, also called voxel carving[1][2][3]. With this method we divide the space the object is in into cubes, called voxels, which are either filled or empty. All voxels are initialized as filled. Then we carve voxels until we are left with just the voxels that together form the object. The carving of the voxels is done using silhouettes of the object created from the frames of the movie. All voxels that lie outside one of these silhouettes are emptied. This way we end up with just the voxels that are part of the object. This is demonstrated in the figure 2. The blue object can be created from the three silhouettes.

For the movies we are using the camera is in a fixed position. The object is positioned on a rotating turntable, giving the camera a view of the object from all sides. The known parameters in our preparation are: camera angle, rotation speed, background color and camera orientation:

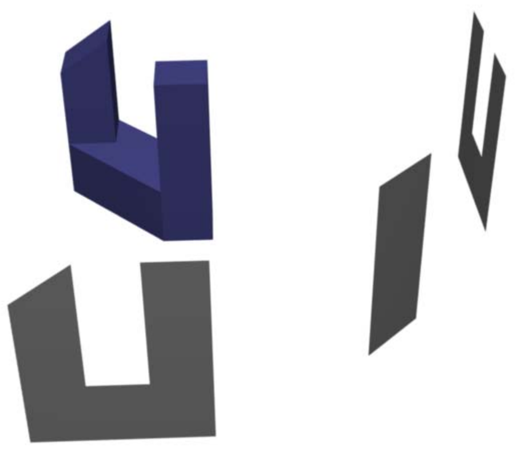


Figure 2. Object creation.

- Camera angle: We know the angle of view of our camera and made sure not to change it for different movies. We need to know this angle because it allows us to determine which voxels are shown on which pixels. Knowing the angle means this part of the process will always be accurate.
- Rotation speed: The turntable rotates at a constant speed, allowing us to easily determine the angle the object has been rotated for every frame of the movie.
- Background color: We need to recognize the pixels on the frames that show the background. We tried to use a background that allows this to be done well.
- Camera orientation: The camera is aimed at the axis of rotation of the turntable and the movies are shot straight from the side, rather than from above. This means we do not have to estimate these parameters, which would result in some inaccuracy.

The photo of our set-up is from our first attempt and thus our first filming scene. Later on we decided to use a background color which we could easily distinct from the colors on the object, as many object have colors similar to the white background seen on the photo. This background also wasn't completely evenly colored. We filmed easy objects with easy shapes and some more difficult objects with more complex shapes and even objects with a color which resembled the background color (slightly different).

Generating the silhouette

To be able to do the sculpting we need a silhouette of the object. We need to know which pixels of the images show the object we are modeling and which show background. Since all the voxels that lie outside of this silhouette will be emptied, we need to make sure that the silhouette we create is never smaller than it should be. If the silhouette is larger than it should, some voxels that should be empty will not be emptied by this frame. This is less of a problem since these voxels may still be emptied by the sculpting of other frames. Some voxels that should be empty can only be recognized as such on one frame and so it is important to create small and accurate silhouettes in order to achieve a high level of detail in our model.

There are many methods to create a silhouette, but because of some aspects of some of the frames in our movies many methods do not generate a good result. One problem we have with some of our images is that the surface of our objects often reflects the background, making it very hard to distinguish between the background and the object. Two examples of this are shown below. We did expect the shiny surface of the ipod in the first image to reflects the background, but as is shown in the second image other objects like this polystyrene shape also reflects the color of the background at certain angles.

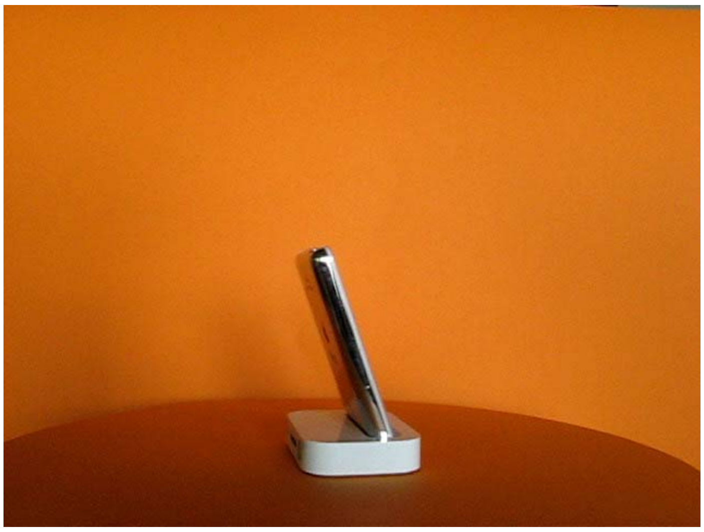


Figure 3. Background reflection.

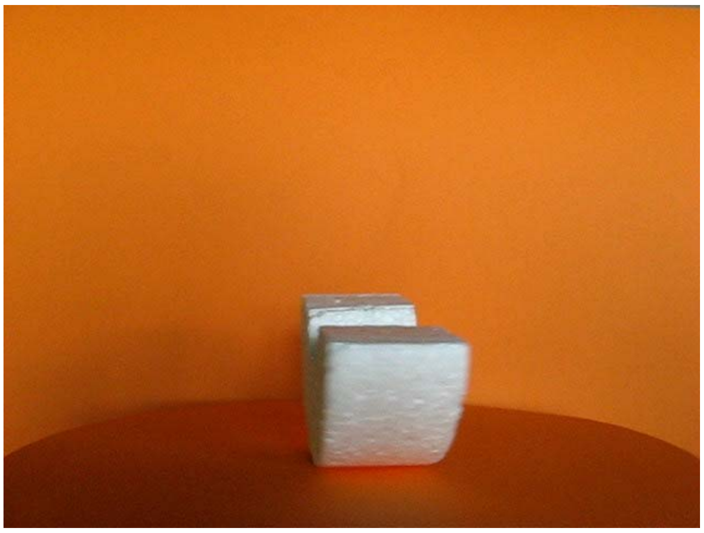


Figure 4. Unexpected background reflection.

Note the many different shades of orange in these images. The flat surface of the turntable is a different color from the background because of the lighting. Around the object there are shadows cast by the object and just below the object the turntable is lighter because of the extra light reflecting from the object. These differences in color could have been averted to some degree by using better lighting, but we decided we were going to try to get some good result with these movies since movies like these are relatively easy to make and it would thus be preferable to be able to use such movies. Note that the colors of the black-and-white images in this section have been inverted.

Gradient

Because of these many different shades of orange in the background and, on some images, very similar colors on the object itself it is not possible to get a silhouette by recognizing the pixels on the background by their color like we had originally planned to do. Fortunately on all of the images in our movies there is a clear edge between the background and the object, even there where the object has turned the color of the background. The colors of the background and part of the object will often be more similar to each other then the different colors of the object, but this does not matter since the background is evenly colored and so every edge on the frames is either between the background and the object or between two colors of the object.
Every pixel that lies between some edges on the image will thus be part of the silhouette.
To find the edges on the image we calculate the gradient. We first attempted calculating the gradient using the intensity of the pixels, which we determined as the root of the sum of the squares of the RGB values. We found that calculating the gradient using this intensity did not work well enough, many edges were not visible on this gradient. Instead we calculate three gradients using the values for red, blue and green. We then build one gradient by using the highest value of those three gradients for each pixel. This gradient shows more edges than a gradient calculated using just intensity. Below an example of a frame with the normal gradient (fig. 6) and the gradient made with our method (fig. 7)

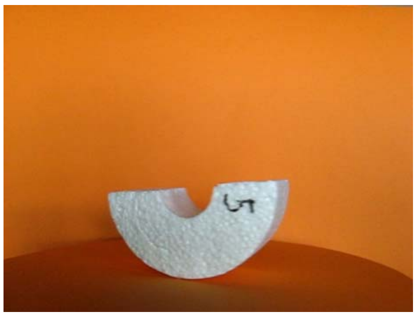


Figure 5. Source image.

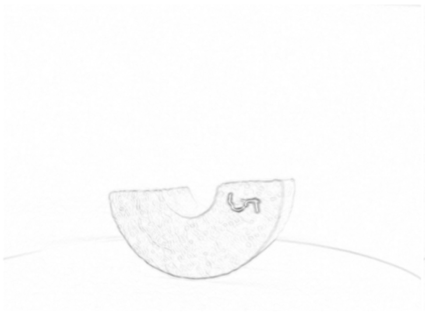


Figure 6. Result with normal gradient method.

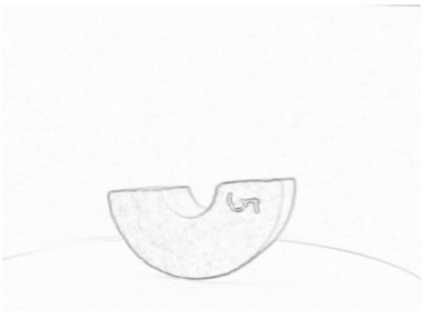


Figure 7. Result with our gradient method.

Non-local-maximum suppression

To find the edges and create an outline from the gradient we tried using the Canny edge detector [4]. We applied non-local-maximum suppression to get thin edges. The intensity of a pixel on the gradient is set to zero if the pixel is not a local maximum. Whether a pixel is a local maximum is determined using the gradient direction. The pixel's intensity needs to be higher than the intensity of two of the pixels on either side of it, which two pixels depends on the gradient direction. If the gradient direction is vertical then the pixel is a maximum if the intensity of the pixels above and below it have a lower intensity. Since there are eight pixels surrounding each pixel the gradient direction is put into one of four ranges, which than determines which two pixels need to have an intensity lower than this pixel in order for it to be a maximum.
Using this method we found some good edges, though there are also some false edges found because of shadows. We did not get a complete outline of the object as there are some openings in it, especially where edges meet. In order to get a silhouette from these lines we need to create a closed contour, which can later be filled to form the silhouette. To get this closed contour we needed an edge linking method. That is, a way to connect the edges to each other until they from a closed outline. We tried doing this using the gradient direction to find the next edge from where one edge ends. This turned out to be problematic because of the false edges from the shadows and because of some edges within the object. Since we need a silhouette generation method that is reliable on over 100 images without creating any silhouettes that are too small we found edge linking to be too unpredictable and so we decided to try another way.

Our silhouette creation method

Using maximums in the gradient to find the exact location of the real edge is useful. Combining this with the assumption that there are no real edges outside of the object we create a method to generate a silhouette:

1. Starting with an empty image we first mark the pixels at the edge of the image as background.
2. Pixels are marked if one of the four pixels next to it is marked and on the gradient the pixel has an higher intensity then the already marked pixel.
3. Repeat until no more pixels are marked.

This way pixels are marked up to the maxima in the gradient, which is where the real edge in the image should be. As with the previous method there are places where there is no useful maximum, for example if there is another edge within the object right after the outer edge in which case the gradient may keep getting higher beyond the real edge. To make this method work we set a limit to the number of times pixels may keep getting higher intensities. With this method the shadows around the object are usually no problem since false edges from shadows are not very sharp and do not end in other edges, the result of this is that pixels will be marked from both sides of the edge of the shadow, marking the whole shadow as background. After finding the right maximum depth for a particular movie a silhouette will be made of every image that is never to small and which is usually not much bigger than it should be.

Implementation

We implemented our way to find the silhouette in the silhouet() function in the testjes class. Creating the silhouette is done in a few steps, most with their own function.
First we use a Gaussian filter [5],[6] to reduce the noise on the image. This makes the image look slightly blurred. This is done by the blur() function. This function uses a 5×5 Gaussian filter with σ = 1.4 to create a new image B from an image A:

$$\mathbf{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}$$

The output of this function for a frame from one of our movies:

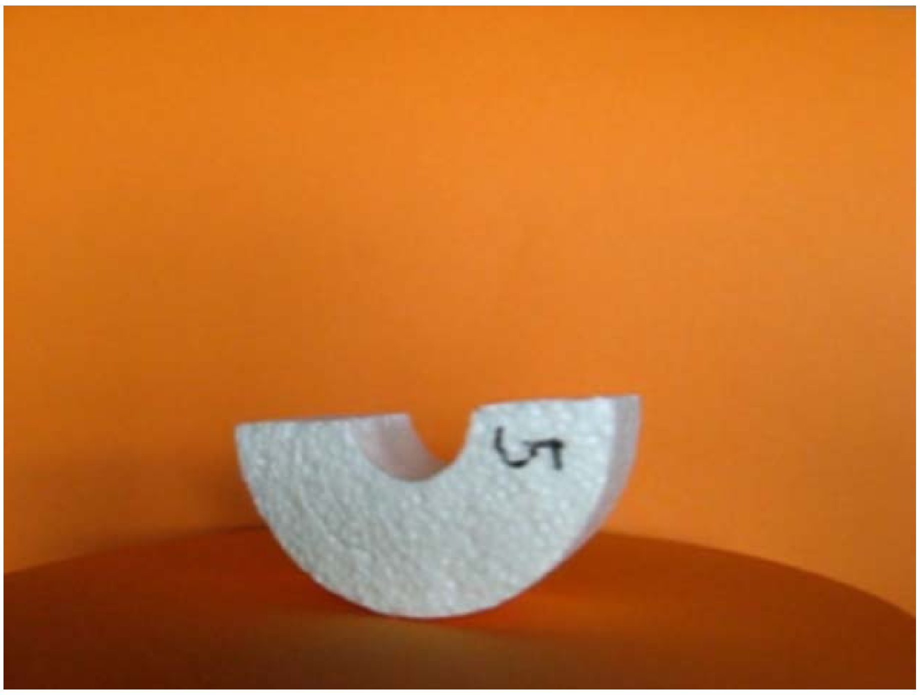


Figure 8. Output of the Gaussian filter.

Then we generate the gradient of this image using the maxrgbgrad() function of the frame class. This function implements the gradient as described earlier. For each pixel it determines three possible values for the gradient using for each of those calculations only the values of one of the three colors red, green and blue. The highest of these three values is then used for the pixel. The function returns a frame with the gradient magnitude but also delivers the gradient direction via one of its parameters.
The output (gradient magnitude) of this function for the same frame:

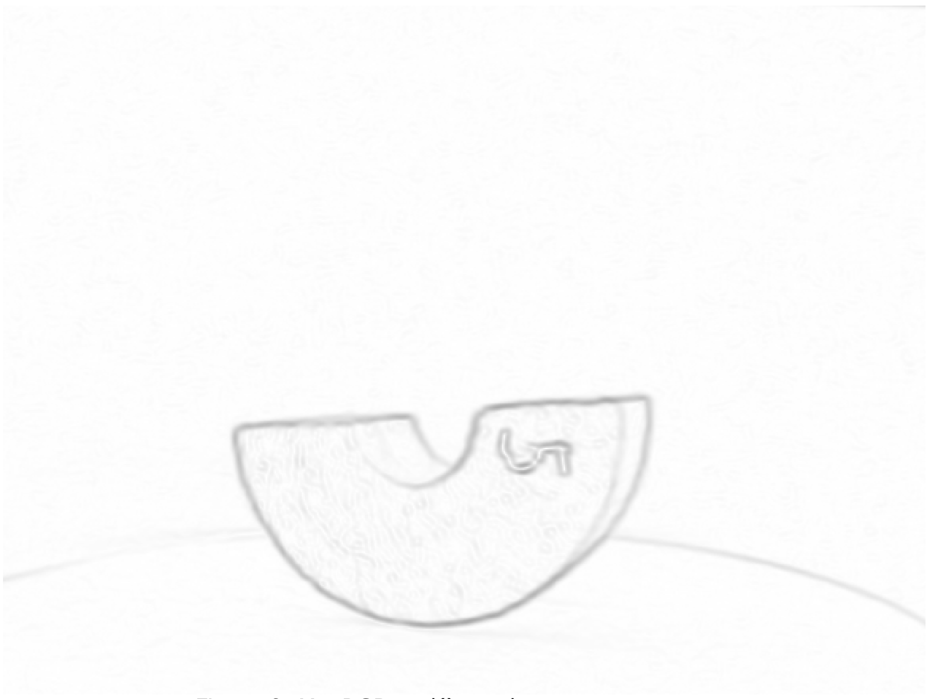


Figure 9. MaxRGBgrad() result.

Getting rid of the turntable edges

In our background there is a big difference in color between the turntable and the back wall which means the outline of the turntable is always visible on the gradient. This line will be marked as background from both sides and will thus not be visible on the silhouette, but close to the object the line may disturb the creation of the silhouette and thus we want to delete a part of this line. This is done by the delbgrand() function of the testjes class. This function takes as its parameters an upper and a lower height limit between which the function sets the intensity of pixels with a high intensity to zero if the gradient direction of the pixel is up or down, with means that the pixel is part of a horizontal line in the gradient. These upper and lower height limits can be set in the interface at the silhouette generation step.
The gradient after this step:

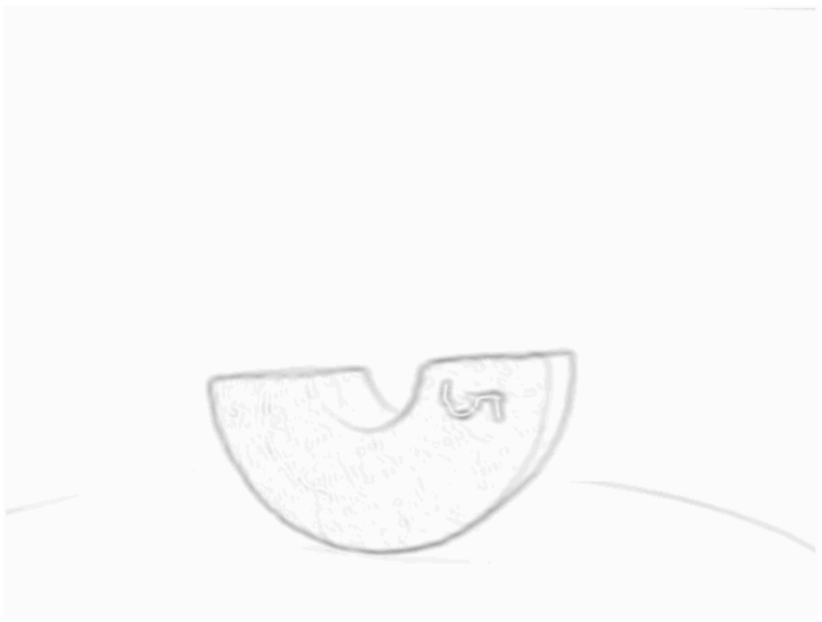


Figure 10. delbgrand() result.

Now we use this image to create the silhouette. This is done in two steps. First we set the intensity of the pixels with an intensity below a certain limit to zero. This limit can be set by the user during the silhouette creation step. This is not done in a separate function but rather in a loop in the silhouet() function. After this the pixels with an intensity greater than zero form an outline of the object. Since the limit needs to be low enough so the pixel with the lowest intensity in the outline is not set to zero there will be many pixels outside of the silhouette that are not set to zero. This means that the outline we get will be bigger than necessary, which will be fixed in the second step of creating the silhouette.
This image below shows all the pixels that are nonzero after this step. Note that the program doesn't usually create this image. The actual image would look exactly like the last one since many nonzero pixels already look like they are completely black in that image.

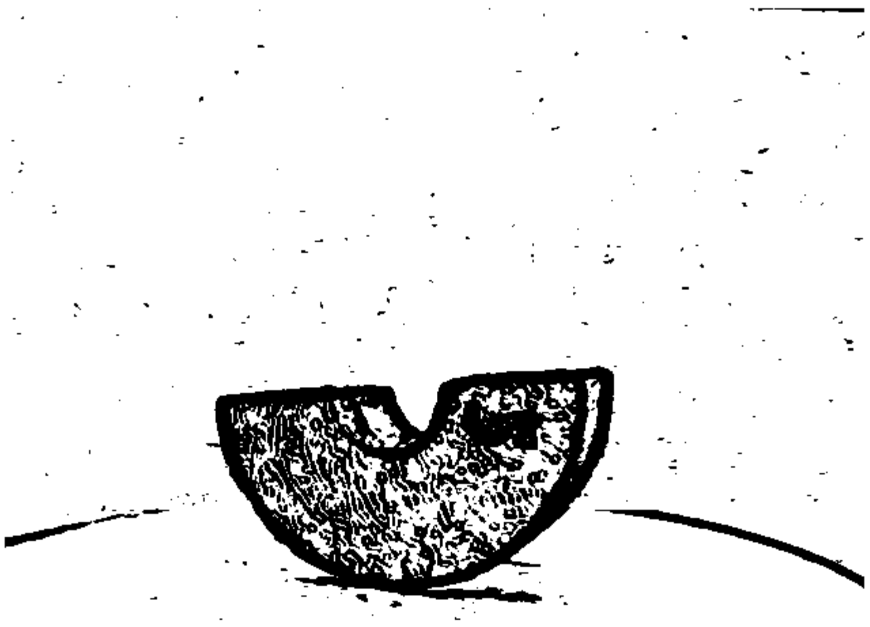


Figure 11. Outline detection.

Now we use the markbg() function to create the proper silhouette. This is done with these steps:

1. The pixels around the edge of the image are marked as background by making the red (R) value nonzero. The green (G) value is set to the maximum number of steps, as set by the user.
2. Every unmarked pixel with one or more marked pixels with G value nonzero next to it is marked if:
   - The pixel's intensity is zero. The G value is copied from the already marked pixel with the highest G value.
   - The pixel's intensity is higher than that of the already marked pixel. The G value is set one lower than that of he already marked pixel with the highest G value.
3. Repeat step 2 until no more change.

This way we mark pixels until we get to a maximum in the gradient or the steps value becomes zero which gives us the silhouette as explained earlier. At the end of the function the red value is inverted and set as all three colors, creating the white silhouette against a black background.

Efficiency

To make markbg() fast we want it to pass through the image only once, instead of iterating until nothing changes. This is achieved by going line by line from top to bottom. On every line we first check pixels from left to right checking whether pixels should be marked because of already marked pixels above, to the left and below the current pixel. Once we get to the end of the line we go back from right to left to check if pixels should be marked because of the pixel to the right of it. Then if pixels on the current line have been marked we go back to the line above this one since the pixels on that line may now be marked from below. On this second pass of that line we will not check the whole line but only the part that has pixels below it that have just been marked. If no pixels are marked on the current line we continue to the next one. The pseudocode for this function:

```
for i from 1 to yres-2 do
   for j from 1 to xres-2 do
      if pixels[i-1][j] marked
         mark pixel[i][j];
      if pixels[i][j-1] marked
         mark pixel[i][j];
      if pixels[i+1][j] marked
         mark pixel[i][j];
   od
   for j from xres-2 to 1 by -1 do
      if pixels[i][j+1] marked
         mark pixel[i][j]
   od
   if pixels have been marked
      i := i-2;
od
```

Below the result from this function:

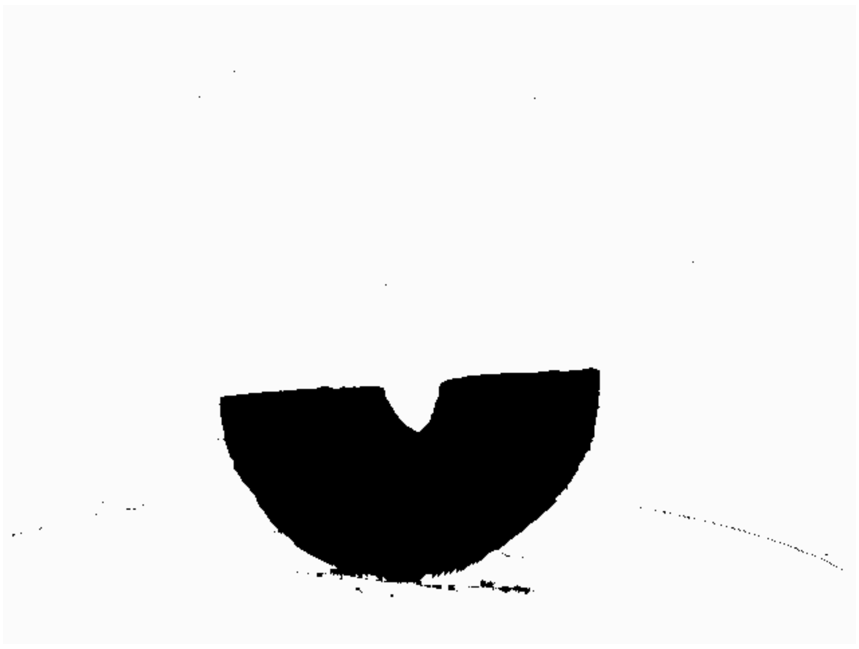


Figure 12. markbg() result.

Two steps remain in the silhouette function once this silhouette has been created. First the delrestjes() function is called to delete the small white areas that are not attached to the silhouette. Thought these are unlikely to cause any problems to the sculpting since they are fairly random and a lot of frames are used for one model, deleting them is very simple and does make the silhouette more accurate.
Finally the silhouette is colored so from this point on we can use only one series of images to sculpt the model and color it. This is done by the frameif() function. To make sure every pixel of the silhouette has a nonzero intensity the pixels on the original image that have an intensity of zero are set to one in the colored silhouette.
The final result of the silhouet() function, note that the black background had been turned white for this image:

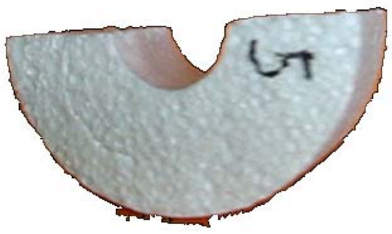


Figure 13. Final result.

Voxel Sculpting

Once the silhouettes have been generated from the frames the next step is sculpting. The sculpting is done in an 640×480×640 bool array called sculpt. Since the images we use have got a resolution of 640×480 this seemed a sensible size for the array as this allows us to sculpt all the detail that may be seen in the images. The sculpting is done in de makeSculpt() function of the testjes class using the silhouettes stored in this testjes object.

Initially all bool values in the array are set to true, all the voxels are filled. We want to empty all the voxels that are seen outside of the silhouette on at least one frame. There are two approaches to doing this, from silhouette to voxels or from voxels to silhouette. The first method means iterating through the frames and for each pixel that lies outside of the silhouette emptying all the voxels that are seen on the pixel. This would be done by going through the voxels along a line through the camera and setting these voxels' value to false.
The second method is iteration through the voxels and determining for each voxel if it is outside of the silhouette on some frame. For each voxel it is determined on which pixel it is shown on each frame. As soon as a frame is found on which this pixel lies outside of the silhouette the voxel is emptied. If all frames are checked and no such frame is found the pixel is left filled.

We first implemented the first method. This turned out to be quite slow as many voxels that are outside of the object lie outside of the silhouette on many frames and this method will empty these voxel several times. Furthermore finding all the voxels shown on a pixel can be hard as further away from the camera there are more voxels on a pixel than close by. The result of this is that if the method is not implemented properly, some far-away voxels may lie 'between' pixels and thus would not be emptied when they should. The second method turned out to be a lot faster.

Rotate in position

During the first step of the program the user defines an series of frames that form a loop. This allows the makeSculpt() function to determine the orientation (and rotation angle) of the object on every frame. The sine and cosine of the angles that the voxels will need to be rotated for every frame are calculated once before the main loop in the function. Since the camera was always aimed at the object from the side rather than from higher up we only need to rotate the voxels around one axis, the y-axis. The main loop in the function iterates through the voxels. Inside this loop the voxel is rotated to determine its location relative to the camera at this point in the movie. We assume that the camera is aimed exactly at the axis of rotation, which we attempted to do while making the movies.

Now that we know where the pixel is relative to the camera we can determine on which pixel the voxel is located. The sculpt array is 640 voxels width and 480 high which is also the resolution of the images. The location of the voxels relative to the camera is such that the center plane fills an entire image. Some of the voxels closer to the camera fall outside the image while further away a plane of voxels does not completely fill an image. We know that the camera we used has an angle of view of about 51.85 degrees. Half the width of the sculpt array is 320 thus we can calculate the distance from the camera to the center plane of voxels (distance d) with:

$\tan(51.8498/2) = 320/d$

We get d = 658.2857 voxels. It is not possible from just the movies to tell how big a voxel is and so we do not know anything about the real size of the model we are about to make, which is not important.

We need to determine the place where the line through the camera and the rotated voxel intersects the central plane of the voxels, Since this central plane fills the entire image the x and y coordinates of this point in 3d space correspond to the x and y coordinates of the pixel on the image. If the point (0,0,0) were in the center of the voxels the x coordinate of the point we're looking for could be calculated as $d/(d+z)*x$ (see image below) with x,z the coordinates of the rotated voxel. With our array the point (0,0,0) is the voxel at the corner, making the code a bit larger than that:

```
step_x = (x-delingx)/((z-320)+cam_d);
step_y = (y-delingy)/((z-320)+cam_d);
x = delingx+step_x*cam_d;
y = delingy+step_y*cam_d;
```

where x, y, z start out as the coordinates of the rotated voxel, cam_d is the camera distance as just calculated and delingx and delingy are the width and height of the sculpt array divided by two.

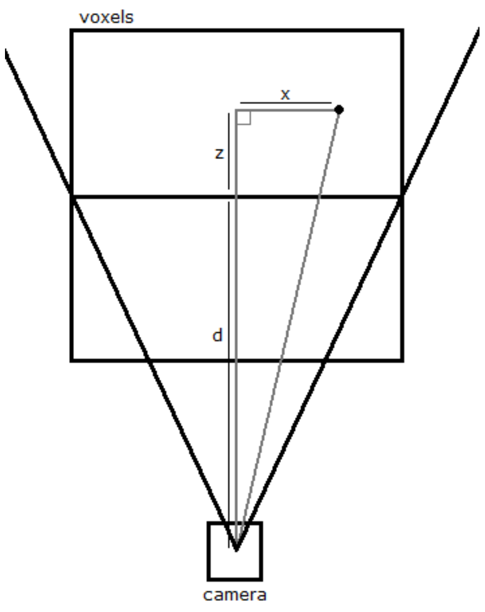


Figure 14. Schematic drawing of our preparation.

With the pixel that the voxel is on now known, the voxel is emptied if this pixel exists (as voxels close to the camera may not be visible on this image) and is not part of the silhouette. If the voxel has been emptied we move onto the next voxel, otherwise we move on to the next frame until the voxel has been checked on all frames and we know the voxel should stay filled.

Coloring the model

After the makesculpt() function has been called we now have the model in the sculpt array. We now want to color the model. Applying colors to a model is often done using texture mapping. With this technique an image is created for the colors which is then mapped to the surface of the model. Since our model is divided into voxels it makes sense that every voxel, and thus every face of the voxel, is one color. This makes texture mapping unnecessary. Instead we define a material for every color, which is applied to the faces of the model that should be that color.
To color the model we use second array, char array sculptcolor with dimensions 640×480×640×3. We find the color of a voxel by determining which frames the voxel is visible on and using the average color of all those frames. Coloring the model is done by the colorsculpt() function of the testjes class, using the frames in the testjes object. Much of the code in this function is similar to the code in the makesculpt() function, in particular the rotating of voxels and finding the pixel that a certain voxel is visible on.

The colorsculpt() function iterates through the sculpt array skipping all the empty spaces and those voxels that have other voxels on all six sides. Having found a voxel that should be colored it is checked first which sides of the voxel are on the outside of the model to determine the exact location in the 3d space that we want to find the color of. This is explained in the figure 15. If only the top of a voxel is visible we want to find the color of that part of the voxel. In fact, the center of the voxel is not visible to the camera and using this location would result in the voxel not getting colored at all.

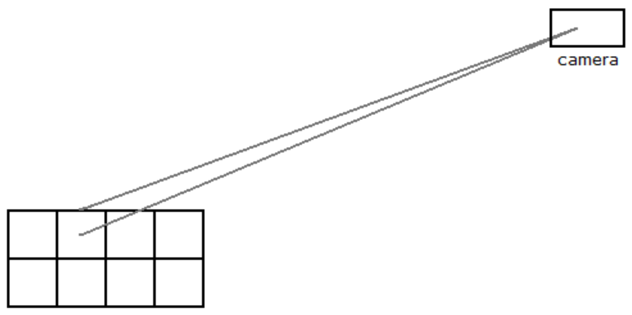


Figure 15. Voxel coloring algorithm illustration.

We want to check if this point is visible from the camera. To do this we first rotate the point to determine its location relative to the camera. We calculate how much the x and y coordinates change when moving one voxel closer to the camera along the z-axis. These steps, including the step in the z direction with length one, are then rotated back so we can move step by step from the original location to the camera, checking whether all the voxels we encounter are empty. If they are the voxel is visible on this frame and we determine on which pixel just like in the makesculpt() function.
Sometimes this pixel will turn out to lie outside the silhouette which is because we do not use the center of the voxel to determine the pixel but instead the side that is on the outside of the model. When this happens we determine the pixel again, this time using the center of the voxel. This pixel will always be within the silhouette as it would otherwise have been emptied during the sculpting.

This is done for every frame and in the end the voxel is given the average of the colors of all the frames on which it is visible. Those voxels that are not visible from any frames are colored a distinctive color to make it clear in the final model that the color of this voxel could not be determined from the movie. The color we use for this has red value 255, blue value 0 and green value 128. This is a distinctive color that is unlikely to appear on an object.

Displaying the model

In our application we display the model using OpenGL.[7] This is done in the class glwidget.[8] Most of the work is done in the makeObject() function. In here the sculpt is copied to the unsigned char figure array. We needed a third state for our optimization. Also we started of using this array to make the sculpt shrink in size. (by tenfold) Later on , after optimization of the number of displayed voxels, we decided the shrinking was not necessarily for our computers. This optimization was done by not giving OpenGL information about the inner voxels. These voxels were given the value 2 by our algorithm which determined if it were inner or outer voxels soo the next algorithm would know which voxels did not need further attention. In OpenGL you can display a quad by specifying: glBegin(GL_QUADS). After that each four vertexes will be recognized as the vertexes of a quad. After that the only thing we need to do is calculate 6 sides x 4 vertexes per side = 24 vertexes and there relative position times the number of voxels. We also gave every vertex a x,y,z-offset so the rotation would be in the center of the object. The offset is equal to the middle of the object. xoffset = blocksize * sizex / 2;

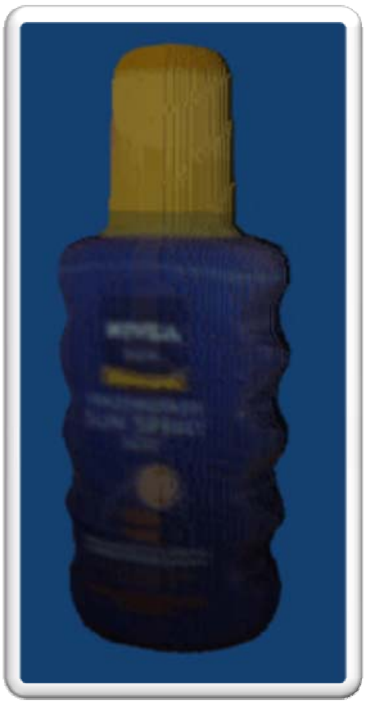


Figure 16. OpenGL result.

Saving the model

The model described in the sculpt and sculptcolor arrays can be saved to a file. This is done in the sculptToFile() function of the testjes class. The file type used is the OBJ file format[9]. This is a simple and universally accepted format. Models in this format are saved as a list of vertices with after that a list of faces which are described by referring to several of the listed vertices, in our case always four. To add the color of the model to the file we use an MTL material file that describes the materials used for the colors of the voxels. In the OBJ file it is specified which material from the MTL file should be used for the faces.

The code is fairly straightforward, we iterate through the voxels and those sides of the voxels that are on the outside of the model are added as faces to the file. We don't want to add four new vertices for every face but instead use any vertices that already exist. To do this we save all vertices we create in an octree. Like these vertices also don't want to create an new material for every voxel as many voxels will have the same color, thus we also use an octree to save the materials in.

Octree

The octree[10] is build from node objects. These node object have an array 'kinds' with eight pointers to other node objects as well as a number and a pointer to an material object to describe a color. The root of the tree describes the whole space. The size of this space can be set when creating an octree object. This space is divided in halves along all three axis, thus it is divided into eight volumes that correspond to the eight node object in the kinds array. Those nodes have also got eight children, dividing the space again in eight parts until you get to a leaf in the tree, which corresponds to a point. When the octree is created it has only a root node. The functions getindex() and getcolor() retrieve the number of an vertex and the material object for a color respectively. They work by means of a recursive function that steps through the tree using the xyz coordinates of the vertex or the RGB value of the color, moving from node to node until arriving at a leaf in the tree. These recursive functions create nodes if they do not yet exist and give new leafs a new number or material object.

In the sculptToFile() function for each voxel that is part of the model we retrieve the material object for the color of the voxel from the octree. If the returned object turns out to be new it is initialized. A material object contains a string 'faces' that contains all the code to describe the faces that use this color. This string will be initialized with a line that states that the following faces will use this material. Material object also has a pointer to the next material object to allow easy iteration through all the materials. Finally we add a declaration of the material to the mtls string which can later be written to the MTL file. When the material object has been retrieved we check the six sides of the voxel to see which are on the outside of the model. For those sides the numbers of the four vertexes are retrieved from the vertex octree. Retrieving these numbers is done via the getvertexindex() function which checks whether the vertex is new and if it is it adds the declaration of the vertex to a string that will later be used to write the first part of the OBJ file, in which the vertexes are declared.
With the numbers of the four vertexes retrieved the code that describes the face is added to the faces string of the material object retrieved earlier.

Decreasing the number of colors

For our movies the colorsculpt() function found several thousand different colors which may give problems when importing the model exported by our program. To fix this the sculptToFile() function has the option of reducing the number of colors to any number required. This is done using k-means clustering.[11]
We want to reduce the number of colors to just k colors. At first the first k colors are used, then for all colors in de list of material object it is determined which of those k colors it is closest to, creating a cluster of similar colors. For all k clusters the mean color is calculated, taking into account the number of faces that use each material. This k means are used as the new k colors and again it is decided for each material which of these colors it is closest to. This is repeated until there is no change or until a limit to the number of iteration is reached. Now we create k new material for the final k colors and iterate through all the old materials again, adding their faces strings to the faces string of the new material closest to it.

Finally we write the OBJ and MTL files. The OBJ file is created by first writing the vertex descriptions string to the file and then iterate through the material objects (either all the material objects or the new k objects depending on whether the number of colors has been reduced) and writing their faces strings to the file.
If we are using all the colors the MTL file is created by simply writing the contents of the mtls string to the file. If the number of colors has been reduced we iterate through the new k material objects and write a material declaration to the file for each of them.

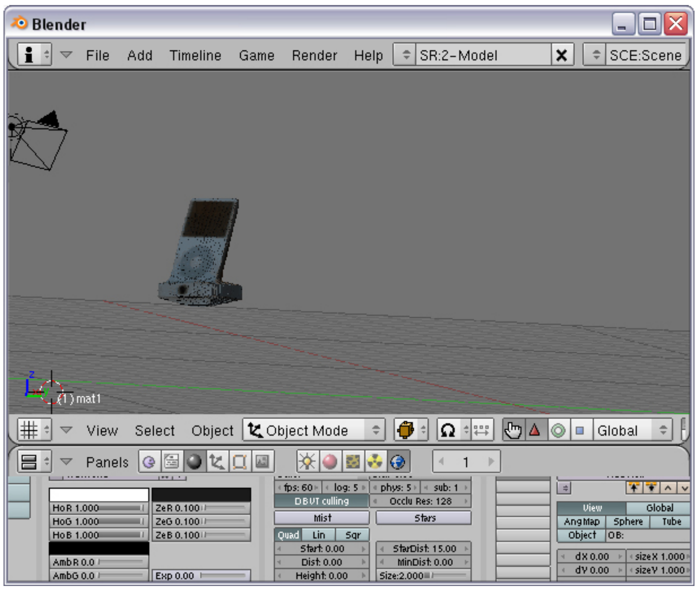

Blender
File Add Timeline Game Render Help
SR:2-Model
SCE:Scene
(1) mat1
View Select Object Object Mode
Global
Panels

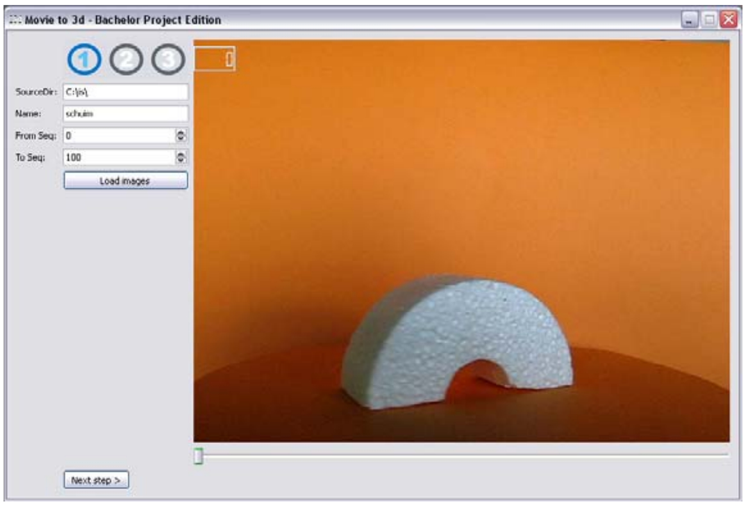
Figure 17. Result in another application (Blender).

Our software

The following software implements the theory above.
We built our application using Qt Creator (with its libraries)[12] and C++.

Step 1

The first layout asks the user to give the location of the image sequence and where the rotation of 360 degrees starts and ends. In this example the images looked at for this particular sequence will be: C:\is\schuim000.png till C:\is\schuim100.png. When the user clicks on "Load images" the images/frames will be loaded in an array. In this array every pixel will be stored in RGB values.

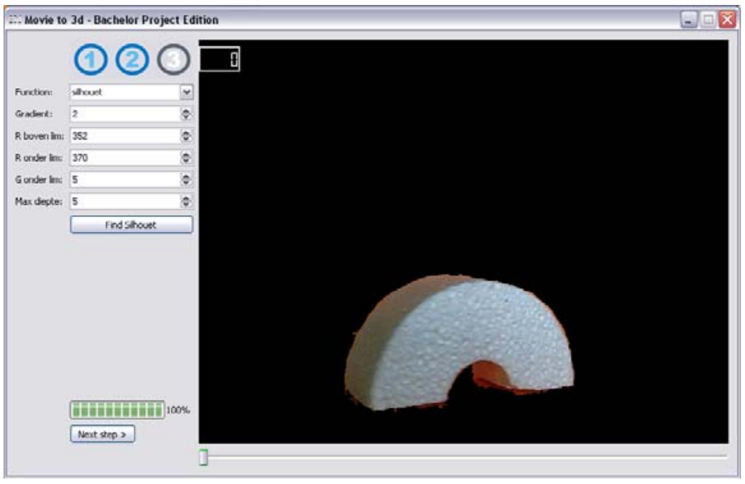


SourceDir: C:\is\
Name: schuim
From Seq: 0
To Seq: 100

Figure 18. First step in our application.

Step 2

The second layout gives the user some options for the application to make its silhouette. "Function" specifies the silhouette function. "Gradient" specifies what gradient will be used. "R boven lim" stands for the upper limit for deleting the horizontal line on the gradient. "R onder lim" stands for the under limit. "G onder lim" for the under limit of the intensity of the gradient. "Max diepte" for the max depth of searching for the max of a gradient.



Function: silhouet
Gradient: 2
R boven lim: 352
R onder lim: 370
G onder lim: 5
Max diepte: 5

Figure 19. Second step in our application.

Step 3

The last step displays the result in a OpenGL screen. The user can rotate the resulting object with its mouse or by using the sliders at the button of the screen. If satisfied, the user can export the model by pressing the "Export Sculpt" button and a saving dialog will appear. The format is compatible with many major applications like: Blender, 3ds Max. The color pink is given to each voxel for which the color could not be determined.

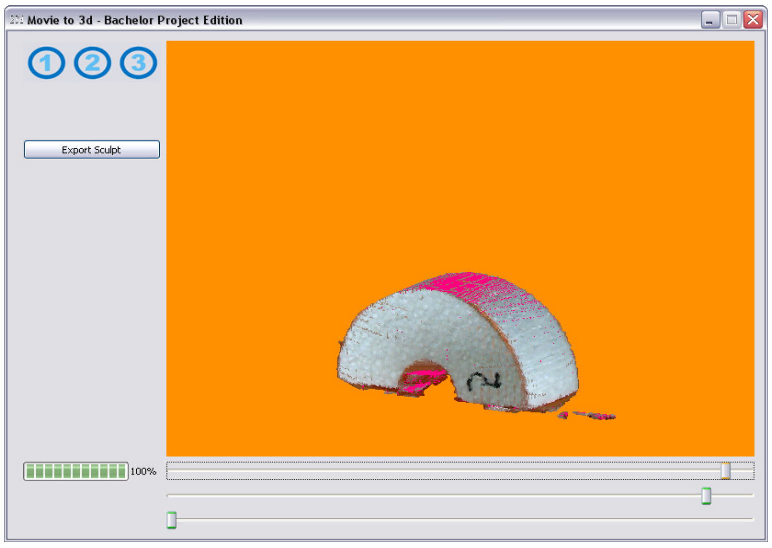


Figure 20. Third step in our application.

Further Research & Improvements

One of the things which could be implemented would be a high definition input sequence. Resulting in HD sculpts. Also a sequence of other camera positions could be used to sculpt and color the object further. The silhouette finding function could be improved by making it able to find holes in the silhouette there where there are openings in the object through which the background can be seen. Also our program cannot detect certain holes in the object. (Fig. 21)

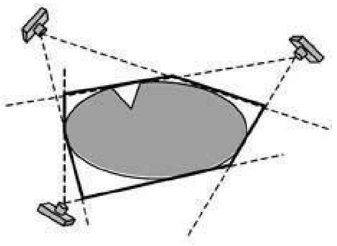


Figure 21. Object with a hole.

Another useful addition would be to make the program in a way that could help 3d modelers with their modeling and provide a tool to make a 3d object out of several sketches.[13]

Looking back on the project

For us it has been a very interesting project. It was the first time for us working with and manipulating images in C++, working with OpenGL and Qt. These experiences can be very useful for future work. There were times that the software did not do what we expected it to do and we got frustrated because of it. (one of the reasons was the not-very-advanced qt debugger and the many hours that would go lost.) However, when we got things right at the end the satisfaction would only be greater. We would like to thank dr. Nies Huijsmans for the guidance and the opportunity to work on this interesting subject. Furthermore we would like to thank mister F.J. Kranenburg (Leiden Institute of Physics) with the help on our practical part of the project.

Conclusion

For our preparation we used our own methods and function. We noticed that when a situation is only a bit different we would need different functionality. It is hard to anticipate on that. For our preparation we tried to obtain the best results as fast as possible with our software. We got some good results with our movies and made sure the user can set certain variables in the algorithms to deal with the differences between our movies, but another preparation would probably need custom code to achieve both speed and good results.

Appendix

Brief explanation of our C++ functions.

Class testjes

Contains most of the functionality, which works with and manipulates the pixel array.

testjes(QString dir, string naami, int fb, int fe, bool HSL);
// constructor, loads images with the name naami in the directory dir from fb to fe, to frame* array frames
~testjes(void);
// destructor, delete all frames in this object
void wiskleur(int b, int l, int r, int o,int lim, int fb, int fe);
// Set all rgb values of the pixels with a certain color to null
void bgpixels(int fb, int fe, int seqlim, int lim);
// find a image from the background by looking what pixels on many frames have the same color
frame * dkleur(int f);
// looks for max R,G,B in a field of 9 pixels after that it attempts to make a silhouette by horizontally and vertically draw lines between the max R,G,Bs found.
void makeSculpt (bool sculpt[640][640][480], int from, int to);
// makes a sculpt of the frames, from "from" till "to". For every voxel in it checks if it is on a background pixel // on a frame somewhere. If so, it deletes the voxel and does not check for this one any more.
frame * setpixels(int k, int fi);
// set for each pixel the value of R,G,B to R,G,B depending on the value of k, used by testjes::maxrgbgrad()
frame * maxrgbgrad(int f, int m);
// uses setpixels and gradient functions to find the maxrgbgrad
frame * markbg(frame * f, int t);
// find a silhouette given a gradient
void sculptToFile(bool sculpt[640][640][480], unsigned char sculptcolor[640][640][480][3], string naam);
// save the sculpt to a .OBJ file
void colorsculpt(bool sculpt[640][640][480], unsigned char sculptcolor[640][640][480][3], int from, int to);
// calculate for every pixel its color
void delbgrand(frame * fm, frame * fd, int b, int o);
// delete the line on the gradient from the background
frame * blur(int f);
// noise reduction, Gaussian filter
frame * silhouet(int f);
// make a silhouette with default parameters
frame * silhouet(int f, int m, int bglijnb, int bglijno, int Ilim, int markbglim);
// creates the silhouet with its parameters
frame * markbg2(frame * f, int t, int d);
// finds the silhouette by deleting everything from the edges until the intensity pixels get lower.
frame * delrestjes(frame * f);
// deletes remainings who are not part of the figure
void frameif(frame * f, frame * iff, bool rand);
// take f from only the pixels who are on iff not black

Class octree

Providing octree functionality to the software

octree(int xresI, int yresI, int zresI);
// constructor, creates a node which contains the whole space
~**octree**();
// destructor, initiates destruction of all nodes in the tree
void recdest(node * n);
// recursively delets all nodes in the octree
material * recgetcolor(node * n, int x, int y, int z, int xn, int yn, int zn);
// search the leaf in the tree which resembles this color and return its material object
material * getcolor(int x, int y, int z);
// initiates the search for the leaf in the tree
int recget(node * n, int x, int y, int z, int xn, int yn, int zn);
// only return the number of the leaf
int getindex(int x, int y, int z);
// initiates search for the leaf with only number as return

Class GLWidget

"Ensures binding between opengl classes and the Qt interface. GLWidget inherits QGLWidget which inherits QWidget it gives to OpenGL the possiblity to integrate itself very nicely in most parts of the application. GLWidget contains an instance of the class Engine." [14]

GLWidget(QWidget *parent = 0);
// The constructor provides default rotation angles for the scene, initializes the variable used for the display list, and sets up some colors for later use.
~GLWidget();
// GLWidget destructor, frees engine
QSize minimumSizeHint() const;
QSize sizeHint() const;
// We provide size hint functions to ensure that the widget is shown at a reasonable size.
void refreshGL(bool sculpt[640][640][480], unsigned char figure[640][640][480], unsigned char sculptcolor[640][640][480][3]);
// refreshes the sculpt displayed by opengl
GLuint makeObject();
// creates a scene of a OpenGL demo block
GLuint makeObject(bool sculpt[640][640][480], unsigned char figure[640][640][480],unsigned char sculptcolor[640][640][480][3]);
// converts the sculpt array to opengl blocks
void setXRotation(int angle);
void setYRotation(int angle);
void setZRotation(int angle);
// the widget provides three slots that enable other components in class to change the orientation of the scene
void initializeGL();
// The initializeGL() function is used to perform useful initialization tasks that are needed to render the 3D scene. These often involve defining colors and materials, enabling and disabling certain rendering flags, and setting other properties used to customize the rendering process.
void paintGL();
// performs painting operations using OpenGL calls
void resizeGL(int width, int height);
// The resizeGL() function is used to ensure that the OpenGL implementation renders the scene onto a viewport that matches the size of the widget, using the correct transformation from 3D coordinates to 2D viewport coordinates.

void mousePressEvent(QMouseEvent *event);
void mouseMoveEvent(QMouseEvent *event);
// These functions simply record the position of the mouse when a button is initially pressed or moved after that

frame class

handles the most elemental methods for our software

bool load(string naam);
// load image in our pixel array
bool load(string naam, bool HSL);
// load image in our array, when HSL=true then HSL colors instead of RGB
bool save(string naam);
// save this frame to a image
QImage arrayToQImage();
// creates a QImage from the array, which enables the software to display it on screen
frame * copy();
// makes a copy of the frame
void hsltorgb ();
// makes a rgb frame out of an hsl frame
frame * gradient (int m);
// makes a gradient of the frame using one of the four methods
frame * gradient (frame * & d, int m);
// like frame::gradient (int m), but also calculates the direction of the gradient
frame * getpart(int b, int l, int r, int o);
// get only a piece of pixel information of the frame
frame * maxrgbgrad(frame * &, int m);
// alternative gradient determination, determine three gradients for each pixel and use the biggest value of those three.

Userinterface class

handles the software its interface and form steps

References

[1] Dyer, Charles R., *Volumetric Scene reconstruction from multiple views,* Foundations of Image Understanding, University of Wisconsin, p. 469-489, 2001.
[2] Felicia Brisc, M.S., *Accelerated Volumetric Reconstruction From Uncalibrated Camera Views*, Dublin City University, July 2006, *http://doras.dcu.ie/157/1/brisc_felicia_2008.pdf*
[3] Kiriakos N. Kutulakos, Steven M. Seitz, *A Theory of Shape by Space Carving, p. 1,* Kluwer Academic Publishers, 2000
[4] Canny, J., *A Computational Approach To Edge Detection*, IEEE Trans. Pattern Analysis and Machine Intelligence, 8:679-714, 1986.
[5] E. Davies, *Machine Vision, Theory, Algorithms and Practicalities*, Academic Press, 1990, p. 42 – 44
[6] R. Gonzalez and R. Woods, *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, p 191.
[7] Khronos Group, OpenGL http://www.opengl.org/sdk/docs/man/, 2009
[8] Nokia Corporation, *QGLWidget documentation*, http://doc.trolltech.com/4.5/qglwidget.html, 2009
[9] Wavefront Technologies, *Object Files (.obj)*, http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/
[10] "SCOREC Mesh Database Users Guide, Version 2.2 - Draft", *SCOREC Report #26-1993*, January 1994
[11] Lloyd., S. P. (1982). *Least squares quantization in PCM. IEEE Transactions on Information Theory **28** (2): 129–137*
[12] Nokia Corporation, *Qt documentation*, http://doc.qtsoftware.com/4.5/index.html, 2009
[13] 1st ASCAAD International Conference, Modeling with gestures: sketching 3d virtual surfaces and objects using hand formation and movements, p.35-40
www.ascaad.org/conference/2004/pdfs/paper3.pdf, december 2004
[14] *http://povama.sourceforge.net/doc/classGLWidget.html*