Cartoon Outlines - Project in High Performance Computer Graphics, EDAN35

Carl Rutholm

Lund University
Sweden

Abstract

This project explored the idea on how to make 3d objects and characters look like cartoons. The goal was to make sure that the objects have well defined outlines as well as definitions to show edges. This was done by implementing two edge detecting algorithms. One uses depth and the other uses normals to determine edges. Using both to complete each other gave good results with thick lines along edges and some definitions.

1 Introduction

For living beings equipped with visual systems, such as humans and human sight, it is no problem to identify shapes and hard edges. Simply by using our senses we can differentiate between a flat, curved and edged shape. For computers however this is very tricky without being armed with the right tools. For a computer that only sees values read from texels and pixels, it sees the different values but not how to interpret them. For this we can provide computers with different algorithms to decide what the values represent. This project explored Sobel and Normal detection algorithms.

2 Algorithms

2.1 Sobel operator

The Sobel operator is an algorithm (sometimes called Sobel Filter) used in computer vision and image processing often for edge detection. The idea is to provide the computer with matrices that it can use to sum up a block of pixels row by row and, by observing the result, determine if its an edge or not. The matrices provided to the computer can vary but two common ones are shown below as sx and sy.

The Sobel operator is based on an image's gray scale values. In this project the distance from camera to object was fetched to be used instead of gray scale values. Since both grays scale and a distance are values it can be used similarly. By grouping a selected pixel and its adjacent pixels into a block (in our case 3x3) the fragment shader can traverse it and perform matrix multiplication with a matrix. This block can be called A. We traverse in x and y direction separately using the corresponding sx and sy matrix. The result for each direction should be a 3x1 matrix called Gx and Gy depending on which way the traversing has been done. By adding all the rows into a final result we can observe and draw conclusions. Ideally the result for summing each row should be 0 if there is no edge and not 0 if there is an edge. However since this is rarely the case with the values a threshold value can be set to let every result that is over the threshold pass as an edge and everything under the threshold pass as not an edge. If it passes as an edge the output color is set to black. If not the color set to the passed in diffuse color. This operation is done on all pixels in the fragment shader before the final image is rendered. See appendix 8 for pseudo code of the implementation.

$$sx = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}.$$

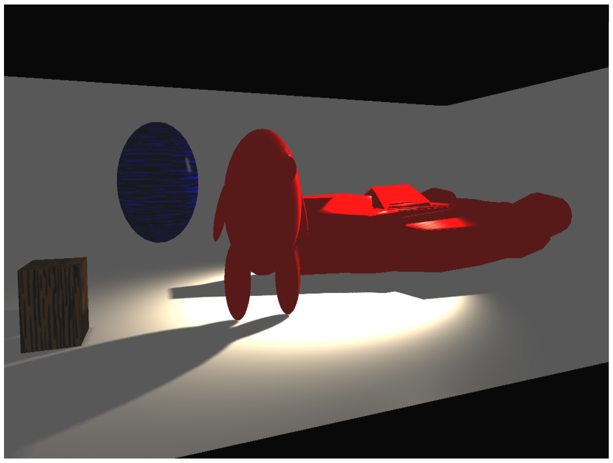
$$sy = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

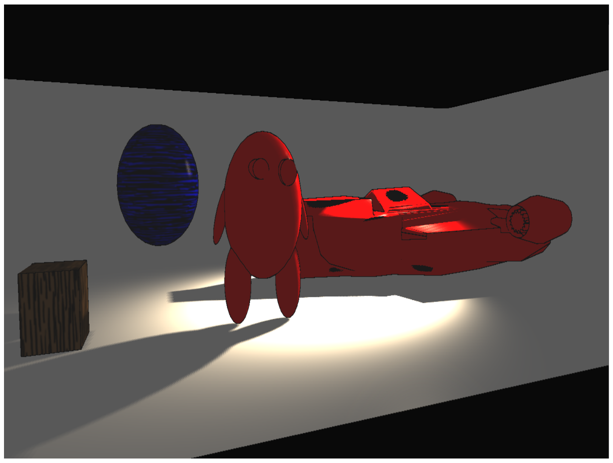
$A =$ fetched 3x3 block of texels
$sx * A = Gx$
$sy * A = Gy$
$G = Gx[0] + Gx[1] + Gx[2] + Gy[0] + Gy[1] + Gy[2]$

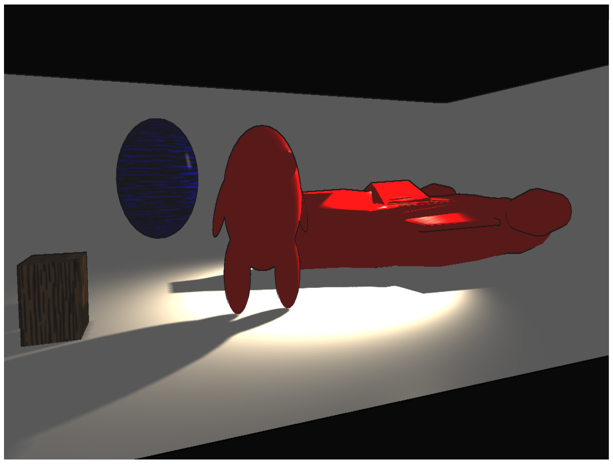
2.2 Normal operator

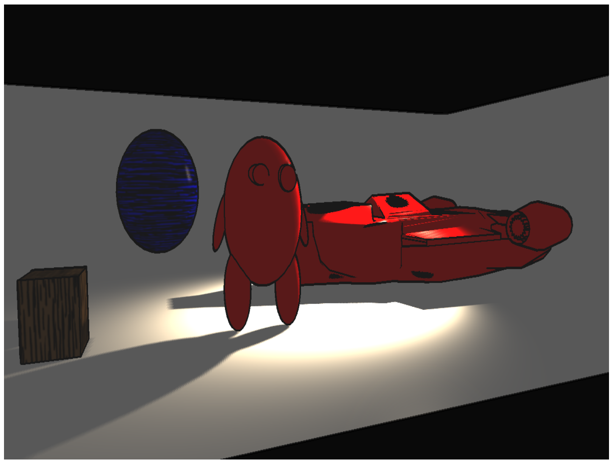
This operator was designed to detect edges by looking at the normal difference on adjacent texels similar to the Sobel Operator. The selected texel's normal is first fetched and then the surrounding texels normals are fetched one by one and compared to the selected texel's normal. If the angle between the compared normals are bigger than a set threshold there is a chance that an edge has been detected. In this case a tracker variable notes it by adding 1 to itself which means that this comparison detected an edge. If the angle ends up not being bigger than the threshold the tracker variable notes this by adding a 0. In the end when all the comparisons has been made in the block, the mean value of the tracker variable is compared to a threshold to output a black or original output color. See appendix 9 for pseudo code of the implementation.

3 Results


Without any operator active


Normal operator active


Sobel operator active


Both Sobel and Normal active

4 Discussion

When observing the results it becomes obvious that for some angles the operators have problems to distinguish an edge and this is mainly because of the thresholds that are hard coded in the shaders. The threshold was experimented on by simply trying different combinations and angles and because of this the thresholds are the biggest factor for errors and unwanted results. Close ups were very pleasing however and showed really good results so camera distance is also something to consider when trying to perfect these operators. If more time was spent on trying to figure out a better solution or experimenting further to optimize the thresholds the results might have differed for the better.

Another factor was the size of the blocks used to compare the depths/normals. If one were to use smaller blocks the precision might have been better but might also suffer from not having enough information about the surroundings. Using too large blocks results in weaker performance and chunkier movements if one is interested in making a controllable experience.

Since I was alone working on this project I could not manage to implement everything I wanted. My plan was to implement more rugged outlines and sketched shadows that uses hatching if I got time over after the main idea was finished. I also wanted to make this into a smaller game where the objective was to put color on all the intractable objects in the scene and to have the color fade in from the players touch all over the object. The objects are intractable right now and the player has simple move controls with a fixed camera as well as a free cam mode. Sadly there was no time to implement this fading coloring method either.

5 Conclusion

The goal for this project was to create well defined outlines and definitions on object using Sobel and Normal edge detecting algorithms. The results shows that this goal was accomplished mostly but with a bit of fine adjustments to perfect it. Observing the different operators individually one can see their strenghts and weaknesses. For instance the normal operator for most scenes and angles since it's looking at the normals but is at the same time a weak option when looking at objects with the same angles and normals. This is where the sobel operator's strength takes over and can compare depths and detect edges on objects with the same normal angle. Both are good in its own but working together gives the best results.

Other than implementing I got experience in the 3D model program Blender where I created the playable character and the platform which I really enjoyed. Further experience and knowledge in OpenGL, shaders, textures and samplers and c++ is also something Im taking with me from this project.

6 Appendix

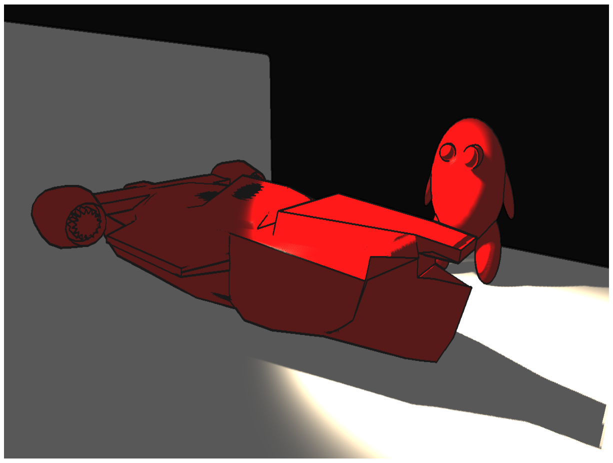


Image 5

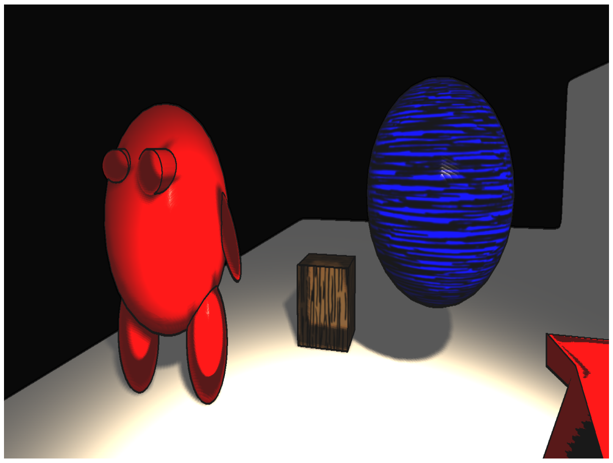
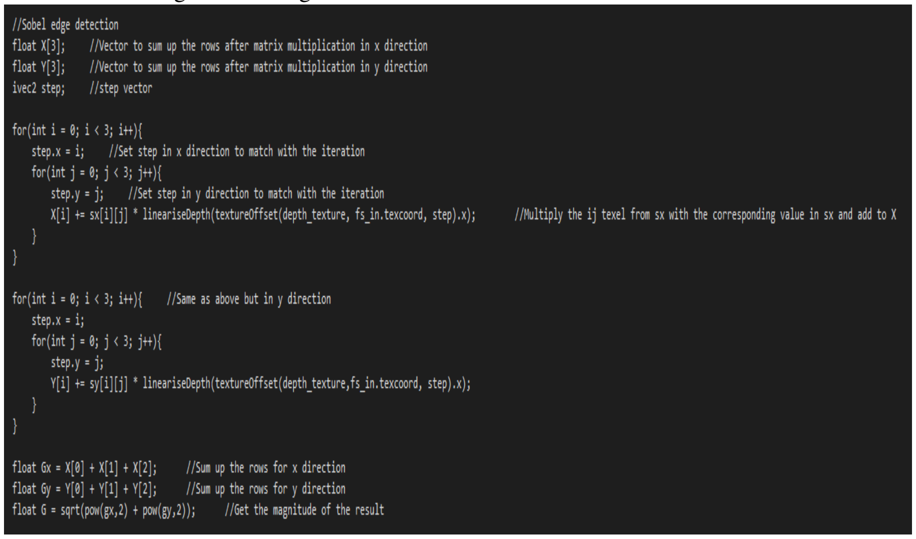


Image 6

Image 7: Creating character in Blender



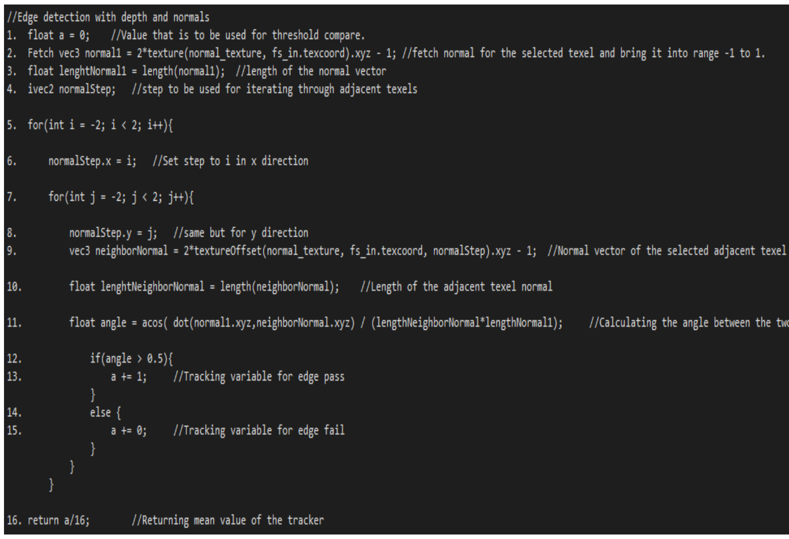
```
//Sobel edge detection
float X[3];    //Vector to sum up the rows after matrix multiplication in x direction
float Y[3];    //Vector to sum up the rows after matrix multiplication in y direction
ivec2 step;    //step vector

for(int i = 0; i < 3; i++){
    step.x = i;     //Set step in x direction to match with the iteration
    for(int j = 0; j < 3; j++){
        step.y = j;     //Set step in y direction to match with the iteration
        X[i] += sx[i][j] * lineariseDepth(textureOffset(depth_texture, fs_in.texcoord, step).x);      //Multiply the ij texel from sx with the corresponding value in sx and add to X
    }
}

for(int i = 0; i < 3; i++){     //Same as above but in y direction
    step.x = i;
    for(int j = 0; j < 3; j++){
        step.y = j;
        Y[i] += sy[i][j] * lineariseDepth(textureOffset(depth_texture,fs_in.texcoord, step).x);
    }
}

float Gx = X[0] + X[1] + X[2];     //Sum up the rows for x direction
float Gy = Y[0] + Y[1] + Y[2];     //Sum up the rows for y direction
float G = sqrt(pow(gx,2) + pow(gy,2));     //Get the magnitude of the result
```

Image 8: Pseudo code for the normal operator



```
//Edge detection with depth and normals
1.  float a = 0;    //Value that is to be used for threshold compare.
2.  Fetch vec3 normal1 = 2*texture(normal_texture, fs_in.texcoord).xyz - 1; //fetch normal for the selected texel and bring it into range -1 to 1.
3.  float lenghtNormal1 = length(normal1);  //length of the normal vector
4.  ivec2 normalStep;   //step to be used for iterating through adjacent texels

5.  for(int i = -2; i < 2; i++){

6.      normalStep.x = i;   //Set step to i in x direction

7.      for(int j = -2; j < 2; j++){

8.          normalStep.y = j;   //same but for y direction
9.          vec3 neighborNormal = 2*textureOffset(normal_texture, fs_in.texcoord, normalStep).xyz - 1;  //Normal vector of the selected adjacent texel

10.         float lenghtNeighborNormal = length(neighborNormal);    //Length of the adjacent texel normal

11.         float angle = acos( dot(normal1.xyz,neighborNormal.xyz) / (lengthNeighborNormal*lengthNormal1);    //Calculating the angle between the tw

12.             if(angle > 0.5){
13.                 a += 1;     //Tracking variable for edge pass
                }
14.             else {
15.                 a += 0;     //Tracking variable for edge fail
                }
            }
        }

16. return a/16;        //Returning mean value of the tracker
```

Image 9: Pseudo code for the sobel operator