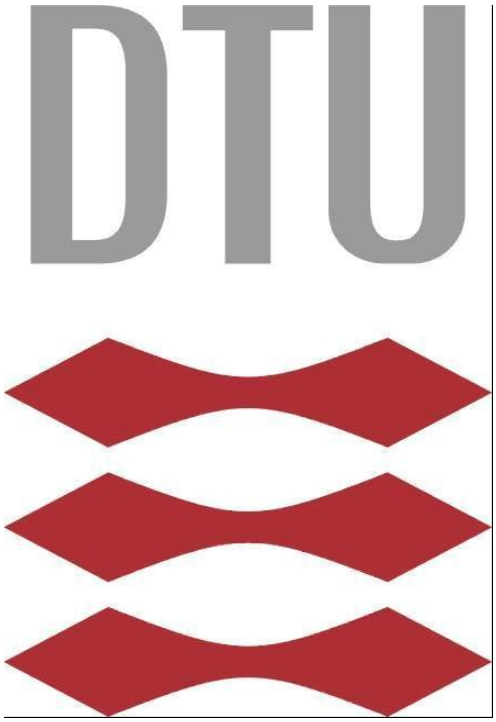
Visually Realistic Tunneling of Volumetric Terrain in Real-Time

Asger Hernández-Reindel



Kongens Lyngby 2012
IMM-B.Eng. - 2012 - 20

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM - B.Eng. - 2012 - 20

This project uses an introductory level understand of OpenGL similar to what is taught but not an explicit part of the 02561 Computer Graphics course at DTU as the Voxel techniques were not part of the curriculum in 2010.
The goal of this thesis is to study and implement a 3D Voxel terrain that can be tunneled through in real-time, and OpenGL will be used to achieve this.
The supervisor is Lector Jakob Andreas Bærentzen from IMM.

Environments of today's world of computer games are generally static. This project tries to find a technique to make environments destructible. The purpose of this project is to demonstrate a technique to produce a dynamic 3D environment for real-time simulation. The idea is to build a volume based 3D Voxel world grid where a changeable polygon mesh surface is drawn. To demonstrate possible functions of this method a mesh surface hill is constructed for creating tunnels.

Content of the report

Chapter 1. Introduction

1.1. Introduction

Normally an environment is built of triangles in a huge flat mesh, but if you tunnel through such an environment you would just create a hole without sides and depth. To make the environment changeable with depth you need a volumetric design.
The technique focused on in this project is Voxels. Voxel is an abbreviation for Volumetric Picture Element, which is a cube or a 3D version of a Pixel. A Voxel representation of terrain might raise some new concerns in 3D computer graphics.
A huge world and terrain would require a lot of Voxels and can hurt performance so much so that real-time rendering would be out of the picture.

1.2. Challenges

The following challenges were setup to be solved during the project

- Creating a Voxel mesh terrain
- Creating a way to destroy part of the terrain
- Creating a surface mesh
- Smoothing the mesh
- Adding attributes such as density to the Voxels

Creating a Voxel mesh terrain:

The first main point of the project is the implementation of a Voxel mesh and the first requirement is to define and find a technique of how to create and store each Voxel from which a full world grid can be built.

Creating a way to destroy part of the terrain:

The second main point of the project is the implementation of a way to destroy Voxels and redraw the world in real-time. A simple remove Voxel function will be created from which more complex system can be built to change the terrain.

Creating a surface mesh:

To render every single existing and non existing Voxel in real-time is expensive from a performance perspective. To improve the performance the drawing function should only concern itself with the surface of the mesh so that only the top visual couple of Voxels are drawn.

Smoothing the mesh:

A Voxel terrain has sharp edges and a method to create a more beautiful looking world could be to implement using a normal Smooth function similar to what is used in anti-aliasing.

Adding attributes such as density to the Voxels:

One of the ideas behind using Voxels is that we can add attributes to each Voxel or each small part of the terrain. The density of a Voxel could be used to determine how brittle the Voxel is, making some parts of the terrain hard like rock and other parts soft like mud.

Chapter 2. Analysis

2.1. A Volumetric Representation

Normally in 3D computer graphics, an environment is built out of triangles in a huge flat mesh. If you tunnel through such an environment you would generally just create a hole without sides or depth. Some techniques exist, such as tessellated distortions, to create the illusion of a small hole dug into the ground, but generally they do not work if you wish to tunnel through the environment. To Achieve this a volumetric design is required.

2.2. Voxels

This can be done by creating the whole environment in small cubes, such that when some cubes in the ground are destroyed, it will be possible to see other cubes under the destroyed cubes. These cubes are usually called Voxels.
Drawing every single cube in a big environment is not cost efficient in performance and can also create a pixelated or square environment. A big upgrade would be to omit drawing any of the voxels and only use them as a way to store the environment information, and then draw a triangle-mesh where the surface of the Voxels is. With this technique the triangle-mesh would change as it is redrawn whenever voxels are destroyed or created in the environment, which means that we would only have to draw the top visual sides of each cube in the surface of Voxels. This technique would allow the use of additional visual improvements to the surface that have been developed in the industry of computer games for decades. Improvements such as smoothing the surface and applying different lighting and texture effects.

2.3. Vots

Another way to implement a volumetric environment could be the use of VOlume doTS, Vots.
Vots are point-based representations of a volume. Instead of using cubes, small spheres are used to construct a 3D model, by giving points or dots a spherical radius size. This implicit representation could be used to create a less pixilated environment, but this technique will most likely have to reinvent decades of 3D computer graphics enhancing techniques that already exist for polygon mesh representations. A future study and comparison of how such enhancing techniques could be converted to be used with a Vots surface could be interesting, but that would be an extensive task and outside the scope of this project.

2.4. A volumetric mesh surface

Drawing every single Voxel in real-time can be expensive, and it is therefore advisable to create a surface mesh only rendering top Voxels. Destroying a Voxel would make deeper voxels behind the destroyed voxel appear and join the mesh surface.
Drawing every single Voxel in a big environment is not cost efficient in performance and can also create a pixelated and not visually pleasing environment. An upgrade could be that we in real-time omit rendering not viewable Voxels such as those hidden behind other voxels hereby creating a voxel surface mesh. A further upgrade would be to modify this surface mesh into a triangle-mesh to create a smooth surface to defeating the 3D pixilation.

2.5. A smooth surface

The technique of creating a smooth triangle surface mesh ais largely used in the entertainment industry. It is therefore a good idea to use a similar technique for the voxel mesh, such that all the techniques already developed to make polygons visually good looking can be reused with this Voxel technique.
Care is required to make the surface not only smooth but also receptive to commonly used graphic enhancement.

Chapter 3. Design

3.1. Design overview

Deciding on the creation of a Voxel representation of terrain that can be tunneled through the following features needs to be designed and implemented:

- Cube drawing functions
- The initialization of the a Voxel grid
- Drawing a hill to tunnel through
- Removal of a Voxel extended by removal of several Voxels
- A surface drawing function
- Creation of a surface mesh
- Addition of density as a Voxel attribute
- smoothing the mesh

3.2. Drawing a Voxel

Drawing a cube in 3D requires positions of each corner point defined in sequentially meaningful order. The world 3D coordinates will be defined by having the y-axis pointing upwards, the x-axis pointing horizontally to the right and the z-axis pointing outwards toward the viewer as seen on the next page in *figure 3.2*. The Voxel or cube is drawn from 8 Corners as an example one side is drawn using the following vertice numbers (look at *figure 3.1*)

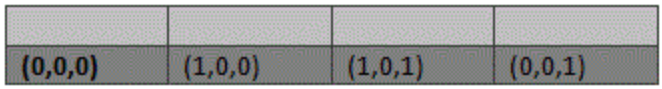


Figure 3.1: Corner order drawing one side of a cube

Figure 3.2 can be used as a reference to these numbers to see that the order of the corners are used clockwise when drawing one side of a cube.

Drawing a full cube requires the construction of the 6 sides in a similar manner and should also be clockwise.

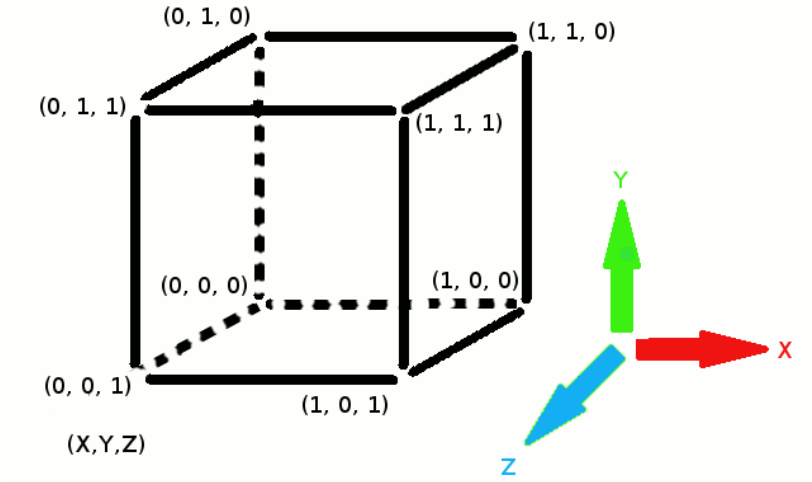


Figure 3.2: Voxel and 3D coordinates

3.2. The voxel grid

The creation of a Voxel terrain is done by drawing cubes that are positioned side by side and stored in memory.

The terrain will be constructed in Voxels and formed as a hill. To achieve this a mathematical formula to construct a hill in 3D is needed. A normal distribution can be used to construct a 2D hill and to add another dimension would simply require the calculations to be performed both for the X direction and the Z direction using the Y direction as the height for the normal distribution.
The normal distribution can be defined as:

$$(1 / \sqrt{2 * \pi * \rho}) * e^{(-1/2) * ((X - \mu)/\rho)^2}$$

3.3. Destroying Voxels

Destroying a Voxel will be done by changing the attribute ‘exist’ of the Voxel in question to zero. This can be done by creating a dot in space and in front of the camera. With a mouse click a distance check can be used to determine whether or not the dot is inside a Voxel that will then be removed.
To extend the method of removing one Voxel to the removal of several Voxels the distance from the centre of a spherical explosion to each Voxel will be calculated by extracting the square root of the distance, where (X,Y,Z) each is set to the power of 2 and added together. The result has to be less than or equal to the radius of the spherical explosion.
To remove voxels in a visually pleasing way we will need a way to evaluate the durability of each Voxel in comparison with the destructive strength of an explosion. This can be done by giving each Voxel a numerical density value from 40 to 100. If this density value goes below one then the material is removed, and the Voxel ‘exist’ attribute will be set to zero. Each Voxel within the range of the explosion will then have its durability reduced by the strength value of the explosion. The strength of the explosion will deteriorate the further the voxel is from the center of the explosion. For example a Voxel near the center of the explosion might get its durability reduced by 100, effectively destroying all voxels near the center, while a Voxel of the outer edge of the explosion might get its durability reduced by 50, meaning that some voxels will be destroyed while others will remain intact, but less durable if another explosion occurs.

3.4. The surface

Originally the terrain is created with all existing Voxels in the shape of a hill. Each Voxel in the grid has an attribute determining whether or not the Voxel exists. During this initialization of the grid an additional attribute can be stored with the Voxel in memory describing if the Voxel is in the surface or not. When everything is redrawn, a check to see if the Voxel is part of the surface will be added to the check of whether or not the Voxel exists. One of the techniques considered and tested will be a smooth triangle mesh based upon the Voxel grid.

Chapter 4. Implementation

4.1. Overview

The GPU computing library OpenGL is used with C++ in Visual Studio 2010 for implementation.
OpenGL was chosen as several computer graphics books have been written on the subject of creating 3D graphics with OpenGL. C++ and Visual Studio were chosen as it is often used in industries that create 3D computer graphics.

The implementation was done step by step in the following order:

- Drawing a Voxel cube
- Creating simplistic camera movement
- Creating a Voxel struct
- Initializing and drawing a large number of Voxels
- Creating a function to remove a single Voxel from the grid
- Removing several Voxel with an explosion-like sphere.
- Creating a Voxel surface mesh
- Creating a triangle mesh based upon the Voxel grid

4.2. Drawing a Cube

Drawing a cube is done by using GL_QUADS with 4 vertices for each side creating a full cube out of 6 such setups each with a different coloured side for easier differentiation between the sides.
The following *figure 4.1* shows an early version of drawing a few cubes

Figure 4.1: Drawing a few Voxels

A simple camera rotation of the whole world was built to evaluate the cubes from different sides, and later the camera movement was evolved into using GLUT_KEY_”arrow-keys” with gluLookAt mouse to rotate up and down, side by side and move forward and backward.
The movement is not optimal but it is sufficient to move around and evaluate the Voxels.

4.3. The Voxel grid

To create the Voxel terrain 3 for-loops inside each other run through each potential Voxel position to create a Voxel and setting up whether or not the Voxels exist and should be drawn:

```
int i,j,k;
for(i=0;i<SIZE_X;i++)
{
	for(j=0;j<SIZE_Y;j++)
	{

		for(k=0;k<SIZE_Z;k++)
		{
			VoxelWorldArray[i][j][k].Exist= round(rand()/(float)RAND_MAX);
		}
	}
}
```

The Voxel array has got all the Voxels stored in memory, and each Voxel element is a C++ struct containing the attributes: ‘exist’, ‘color’ and ‘density’. ‘exist’ tells us if a Voxel will be drawn, ‘color’ decides the color of the Voxel and ‘density’ is a number representing the strength of material the specific Voxel is made of.
In further expansions of the structure Voxel gets another attribute: ‘isSurface’.
This attribute is added to determine whether or not the Voxel is part of the surface which will be used to find out if the Voxel will be drawn.

The following *figure 4.2* shows an older iteration of what the Voxel terrain looked like:

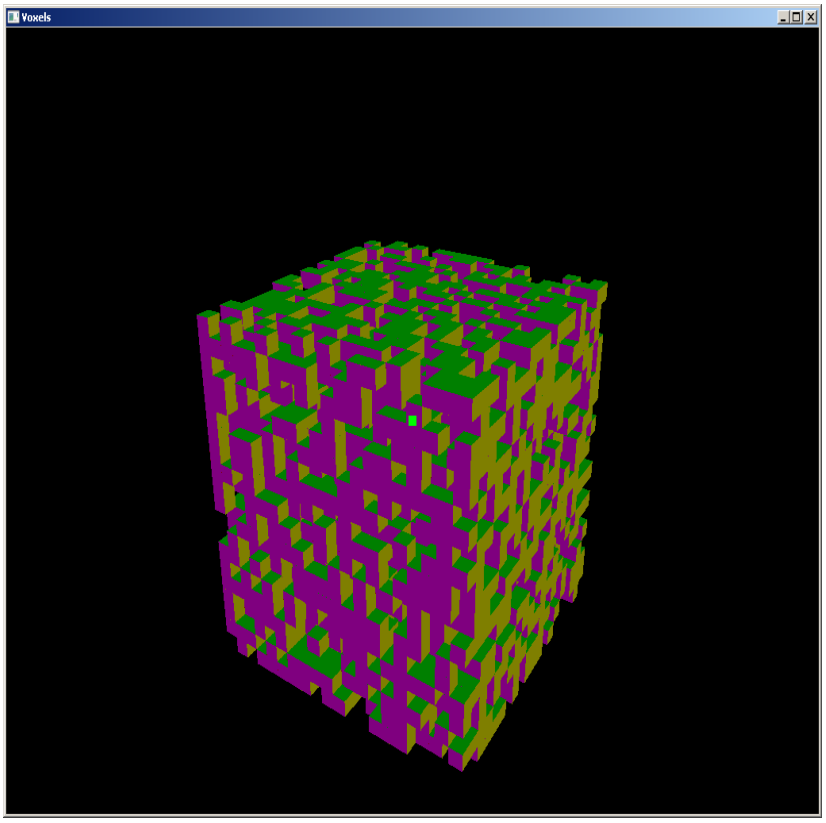


Figure 4.2: Drawing a lot of Voxels at random

In the code the function to draw this version is called InitWorldV1() and can be uncommented in the Main function while commenting out the InitWorld().

The '**VoxelWorldOld.avi**' video of a dig through this Voxel grid can be found digitally on the CD or campusnet group.

Using the equation for a normal distribution, as mentioned in the design part of this report, a hill is constructed in the same way as the old Voxel grid was created. Look at *figure 4.3*

4.4. Destroy Voxels

A reference point is needed to destroy a Voxel, and by placing a GL_POINT one unit in front of the camera we get a point from which we can check if the point is inside a Voxel.
After a left-button mouse click the camera's position added to the camera's direction is used in a variation of pythagoras theorem to calculate the distance from the GL_POINT to the specific Voxel starting point, and this distance only has to be smaller than the radius of the explosion to decide if the Voxel is removed. Removing a Voxel is done by setting the specific Voxel's 'exist' attribute to zero.
Figure 4.3 shows how the Voxels currently looks in a hill formation:

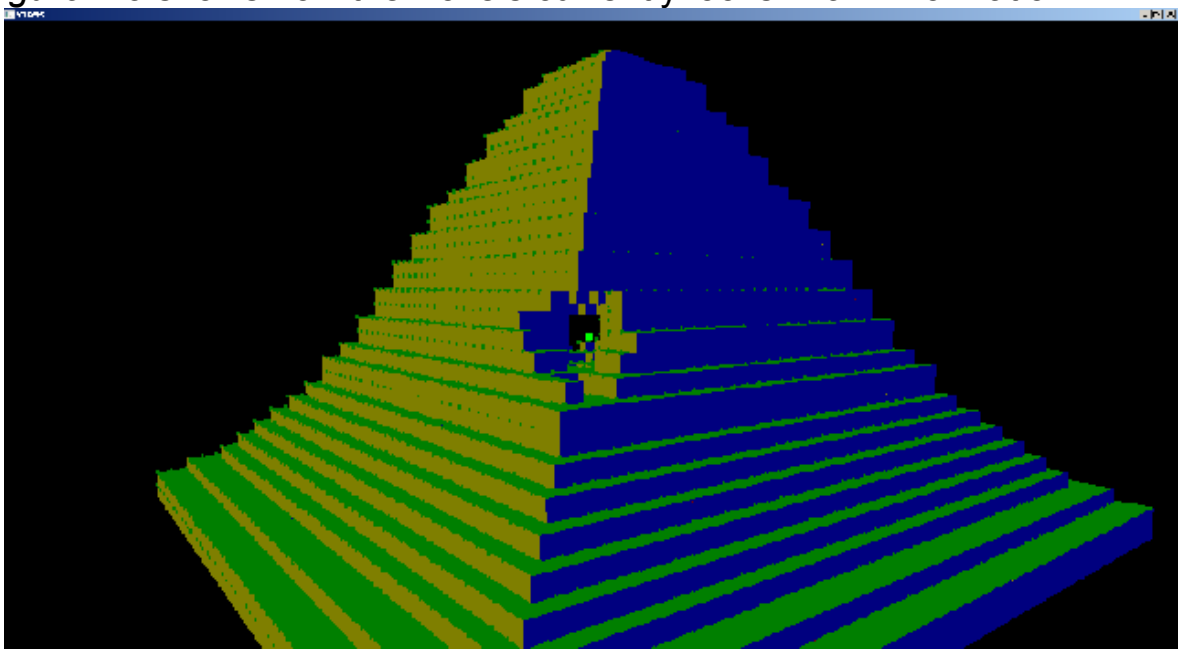


Figure 4.3 Voxel hill tunneled through with spherical explosions.

The 'VoxelWorld.avi' video of a dig through a Voxel hill can be found digitally on the CD or campusnet group. The tiny dark-green points are each Voxel's starting point.

4.5. Triangle surface

To create a triangle surface a new structure “Triangle” was created with float vertex attributes for each of a triangle's three corners, a ‘color’ attribute and a ‘normal’ attribute.
Then we loop through each Voxel checking for ‘isSurface’ and draw a triangle from the center of a Voxel to create a smoother surface.

On *figure 4.4* of the triangle surface mesh it can be seen that the mesh begins from the center of the Voxels.

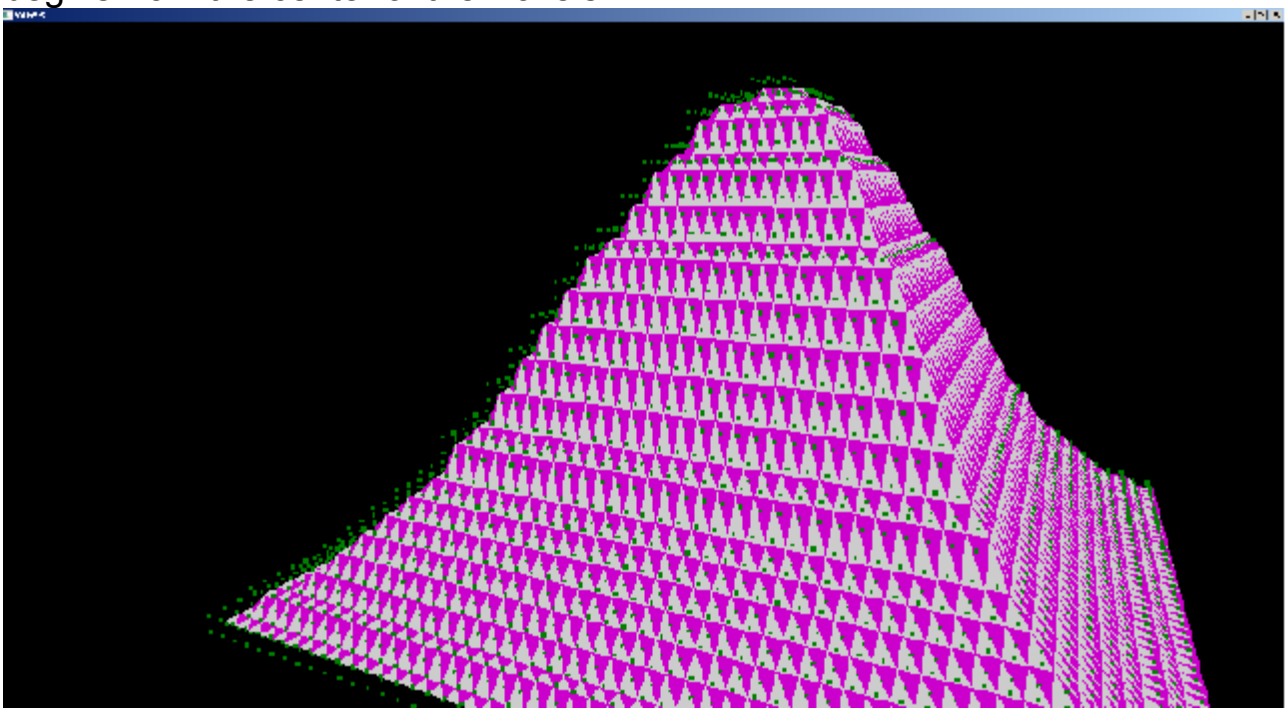


figure 4.4: Triangle mesh and dark-green starting points of each voxel

This can been seen by the fact that the triangle mesh begins one unit to the right of each Voxel’s starting point. To the left we can see green points in space away from the mesh while to the right the green starting points are barely visual.

Chapter 5. Final Discussions

5.1. Discussion

This chapter in the report contains evaluations and thoughts on the study of the project, methods implemented, further studies to look into and a conclusion of the project.

5.2. Inspiration for the project

It is no secret that this project has been inspired by the independently developed very successful game Minecraft.

Figure 5.1: The computer game: Minecraft (link to image in reference list)

The world terrain and objects are procedurally created in a massive Voxel Grid, and the game is all about mining and digging into the terrain to

gather resources from which one can build houses, bridges, railroads and much more.
The author of this report wanted to create a way to construct a changeable Voxel terrain and try to evaluate if a technique using Voxels could be used to create a smooth terrain similar to what is found on modern high-end graphically games.

5.3. Voxel Engines and editors

This project might not have been a very successful in proving that today's games can adapt to using Voxel terrain or Voxel objects to create interesting and realistic looking destructible terrain and objects using a subtractive Voxel method, but today several computer graphic engines and editors are bragging about their Voxel terrain with subtractive and additive Voxel functions built into them.
Here is a two real-time Voxel environments:

Figure 5.2 Voxel terrain from The C4 Engine\Figure 5.3 Voxel terrain game on a nokia phone

While this project never reached a conclusive proof that Voxels can be used in high-end games, it does seem that we are getting closer if we are not already there.

5.4. Conclusion and recommendations for further study

The most basic version of this project was a success. A Voxel terrain in the form of a hill was created, and a Voxel removal function showed that tunnelling through terrain in 3D real-time is possible. Creating a smooth surface from the Voxels caused issues when trying to manipulate the triangle surface for tunneling. A better way to implement a triangle surface than what was done in this project would have been to create the triangle mesh directly on top of the Voxel-surface and then applying a smooth function to this triangle mesh rather than trying to draw the triangle mesh smooth right from the start. It would also be relevant to study the type of performance that can be gained by using techniques like ‘marching cubes’ or ‘sparse Voxel Octrees’.

figure 5.4: An animated character created with sparse Voxel Octrees

However this report and project of visually realistic tunneling through volumetric terrain has not been finished satisfactorily, and while a lot of work has been put into this project, a true conclusion would require a new study and project.

Bibliography

[1] Interactive Computer Graphics: A top-down approach using OpenGL, Edward Angel, 2009-5th edition

[2] OpenGL: A Primer, Edward Angel, 2007-3rd Edition

[2] VOTS: VOlume doTS as a Point-Based Representation of Volumetric Data, Sören Grimm, Stefan Bruckner, Armin Kanitsar, Eduard Gröller, 2004
.
[3] Realtime Rendering of Voxel-based Scenes, Frederik P. Aalund,
Søren V. Poulsen, Jeppe U. Walther, 2011

[4] Animated Sparse Voxel Octrees, Dennis Bautembach, February 25, 2011
[4] figure 5.3

Figure 5.1: http://www.gamereactor.dk/media/21/minecraftmyheart_232155.jpg

Figure 5.2: http://www.terathon.com/wiki/images/c/c9/Terrainarch.jpg

Figure 5.3: http://i.ytimg.com/vi/yviFmj0lq4A/0.jpg

Figure 5.4: http://bautembach.de/wordpress/wp-content/uploads/imrod_walk_cycle.jpg

APPENDIX A: Code

3Dproject.cpp

```
1.  #include <stdlib.h>
2.  #include <stdio.h>
3.  #include <string.h>
4.  #include <GL/glut.h>
5.  #include "VoxelStruct.h"
6.  #include <math.h>
7.
8.  #define mouseSpeed 0.01f
9.  #define keySpeed 0.5f
10. #define SIZE_X 50
11. #define SIZE_Y 50
12. #define SIZE_Z 50
13. #define SCALE 0.4
14. #define SIZE_TRI 2000000
15.
16. const double PI = 3.141592;
17. float camPosX=0.0, camPosY=0.0, camPosZ=0.0;
18. float camDirX=0.0, camDirY=0.0, camDirZ=1.0;
19. float camUpX=0.0, camUpY=1.0, camUpZ=0.0;
20. float mouseDeltaX=0.0, mouseDeltaY=0.0;
21. int pressedKeyUp=0, pressedKeyDown=0, pressedKeyLeft=0, pressedKeyRight=0;
22. int triangleAmount=0;
23.
24. int windowSizeX=512, windowSizeY=384;
25. float mouseCenterX = windowSizeX/2.0 , mouseCenterY = windowSizeY/2.0;
26.
27. //Camera: Where zero equals keys not being pressed.
28. float deltaMove = 0;
29. //Vectors representing the camera's direction
30. float lx=0.0,lz=-1.0, ly=0.0;
31. //XZ position of the camera
32. float x=0.0, z=5.0, y=1.0;
33.
34. #define round(i) (i < 0.5 ? 0.0 : 1.0)
35.
36. Voxel VoxelWorldArray[SIZE_X][SIZE_Y][SIZE_Z];
37. Triangle TriangleSurfaceArray[SIZE_TRI];
38.
39. float calc3DotProd(float x1, float x2, float y1, float y2, float z1, float z2)
40. {
41.     return x1*x2 +y1*y2+z1*z2;
42. }
43.
44. //en.wikipedia.org/wiki/Cross_product
```

```
45. float calcCrossProdX(float a2, float b3, float a3, float b2)
46. {
47.     return a2*b3-a3*b2;
48. }
49. float calcCrossProdY(float a3, float b1, float a1, float b3)
50. {
51.     return a3*b1-a1*b3;
52. }
53. float calcCrossProdZ(float a1, float b2, float a2, float b1)
54. {
55.     return a1*b2-a2*b1;
56. }
57.
58. void destroyVoxel(int x,int y,int z)
59. {
60.     float scaleX = x*SCALE, scaleY =y*SCALE, scaleZ =z*SCALE;
61.     VoxelWorldArray[x][y][z].exist = 0;
62.     for(int i=0; i<triangleAmount; i++)
63.     {
64.         if(TriangleSurfaceArray[i].cornerA[0] == scaleX &&
65.             TriangleSurfaceArray[i].cornerA[1] == scaleY &&
66.             TriangleSurfaceArray[i].cornerA[2] == scaleZ)
67.         {
68.             TriangleSurfaceArray[i].cornerA[1] -= SCALE;
69.         }
70.         if(TriangleSurfaceArray[i].cornerB[0] == scaleX &&
71.             TriangleSurfaceArray[i].cornerB[1] == scaleY &&
72.             TriangleSurfaceArray[i].cornerB[2] == scaleZ)
73.         {
74.             TriangleSurfaceArray[i].cornerB[1] -= SCALE;
75.         }
76.         if(TriangleSurfaceArray[i].cornerC[0] == scaleX &&
77.             TriangleSurfaceArray[i].cornerC[1] == scaleY &&
78.             TriangleSurfaceArray[i].cornerC[2] == scaleZ)
79.         {
80.             TriangleSurfaceArray[i].cornerC[1] -= SCALE;
81.         }
82.     }
83. }
84.
85. void checkHitVoxel(float x, float y, float z)
86. {
87.     float distX, distY, distZ, distXYZ;
88.     int i,j,k;
89.     int RADIUS = 3;
90.     for(i=0;i<SIZE_X;i++)
91.     {
92.         for(j=0;j<SIZE_Y;j++)
93.         {
94.             for(k=0;k<SIZE_Z;k++)
95.             {
```

```
96.             if (VoxelWorldArray[i][j][k].exist)
97.             {
98.                 distX = abs(i*SCALE-x);
99.                 distY = abs(j*SCALE-y);
100.                        distZ = abs(k*SCALE-z);
101.
102.                        distXYZ = sqrtf(distX*distX + distY*distY + distZ*distZ);
103.                        if(distXYZ <= SCALE*RADIUS)
104.                            destroyVoxel(i,j,k);
105.                        /*//remove one voxel
106.                        if(distX <= 0.5*SCALE && distY <= 0.5*SCALE && distZ
    <=0.5*SCALE)
107.                        {
108.                            destroyVoxel(i,j,k);
109.                        }*/
110.                    }
111.                }
112.            }
113.        }
114.    }
115.
116.    void initTriSurface(void)
117.    {
118.        int heightB, heightC, heightD;
119.        int i, j, k, triCounter =0;
120.        for(i=0;i<SIZE_X-1;i++)
121.        {
122.            for(j=0;j<SIZE_Y;j++)
123.            {
124.                for(k=0;k<SIZE_Z-1;k++)
125.                {
126.                    if(VoxelWorldArray[i][j][k].isSurface)
127.                    {
128.                        for(int m=0; m<SIZE_Y; m++)
129.                        {
130.                            if(VoxelWorldArray[i+1][m+0][k+0].isSurface)
131.                                heightB= m;
132.                            if(VoxelWorldArray[i+1][m+0][k+1].isSurface)
133.                                heightC= m;
134.                            if(VoxelWorldArray[i+0][m+0][k+1].isSurface)
135.                                heightD= m;
136.                        }
137.                        //First Triangle
138.                        //Grey
139.                        TriangleSurfaceArray[triCounter].color[0] = 0.8;
140.                        TriangleSurfaceArray[triCounter].color[1] = 0.8;
141.                        TriangleSurfaceArray[triCounter].color[2] = 0.8;
142.                        TriangleSurfaceArray[triCounter].color[3] = 1;
143.                        //A 000
144.                        TriangleSurfaceArray[triCounter].cornerA[0] = (i+0)*SCALE;
145.                        TriangleSurfaceArray[triCounter].cornerA[1] = (j+0)*SCALE;
```

```
146.                    TriangleSurfaceArray[triCounter].cornerA[2] = (k+0)*SCALE;
147.                    //B 100
148.                    TriangleSurfaceArray[triCounter].cornerB[0] = (i+1)*SCALE;
149.                    TriangleSurfaceArray[triCounter].cornerB[1] = (heightB)*SCALE;
150.                    TriangleSurfaceArray[triCounter].cornerB[2] = (k+0)*SCALE;
151.                    //C 101
152.                    TriangleSurfaceArray[triCounter].cornerC[0] = (i+1)*SCALE;
153.                    TriangleSurfaceArray[triCounter].cornerC[1] = (heightC)*SCALE;
154.                    TriangleSurfaceArray[triCounter].cornerC[2] = (k+1)*SCALE;
155.
156.                    TriangleSurfaceArray[triCounter].normal[0] = 0;
157.                    TriangleSurfaceArray[triCounter].normal[1] = 1;
158.                    TriangleSurfaceArray[triCounter].normal[2] = 0;
159.                    triCounter +=1;
160.
161.                    //Secound Triangle
162.                    //Purple
163.                    TriangleSurfaceArray[triCounter].color[0] = 0.8;
164.                    TriangleSurfaceArray[triCounter].color[1] = 0;
165.                    TriangleSurfaceArray[triCounter].color[2] = 0.8;
166.                    TriangleSurfaceArray[triCounter].color[3] = 1;
167.                    //A 000
168.                    TriangleSurfaceArray[triCounter].cornerA[0] = (i+0)*SCALE;
169.                    TriangleSurfaceArray[triCounter].cornerA[1] = (j+0)*SCALE;
170.                    TriangleSurfaceArray[triCounter].cornerA[2] = (k+0)*SCALE;
171.                    // 101
172.                    TriangleSurfaceArray[triCounter].cornerB[0] = (i+1)*SCALE;
173.                    TriangleSurfaceArray[triCounter].cornerB[1] = (heightC)*SCALE;
174.                    TriangleSurfaceArray[triCounter].cornerB[2] = (k+1)*SCALE;
175.                    // 001
176.                    TriangleSurfaceArray[triCounter].cornerC[0] = (i+0)*SCALE;
177.                    TriangleSurfaceArray[triCounter].cornerC[1] = (heightD)*SCALE;
178.                    TriangleSurfaceArray[triCounter].cornerC[2] = (k+1)*SCALE;
179.
180.                    TriangleSurfaceArray[triCounter].normal[0] = 0;
181.                    TriangleSurfaceArray[triCounter].normal[1] = 1;
182.                    TriangleSurfaceArray[triCounter].normal[2] = 0;
183.                    triCounter +=1;
184.
185.                }
186.            }
187.        }
188.    }
189.    triangleAmount = triCounter;
190. }
191.
192. void InitWorld(void)
193. {
194.    int i,j,k, x, z;
195.    float height_x, height_z, μ, RHO;
196.    μ = SIZE_X/2.0;
```

```
197.            RHO = 0.2;
198.            //create voxel hill
199.            for(i=0;i<SIZE_X;i++)
200.            {
201.               for(j=0;j<SIZE_Y;j++)
202.               {
203.                  for(k=0;k<SIZE_Z;k++)
204.                  {
205.                     x = i;
206.                     z = k;
207.                     //initialize a hill
208.                     //normal distribution : (1./(sqrt(2.*3.14*RHO)))*exp(-1.*(pow(X - μ, 2)/
    (2.*pow(RHO, 2))))
209.                     height_x = (1./(sqrt(2.*3.14*SIZE_X*RHO)))*exp(-1.*(pow(x - μ, 2)/
    (2.*pow(SIZE_X*RHO, 2))))*SIZE_Y*5;
210.                     height_z = (1./(sqrt(2.*3.14*SIZE_Z*RHO)))*exp(-1.*(pow(z - μ, 2)/
    (2.*pow(SIZE_Z*RHO, 2))))*SIZE_Y*5;
211.                     if( j <= height_x && j <= height_z)
212.                        VoxelWorldArray[i][j][k].exist= 1;
213.                  }
214.               }
215.            }
216.
217.            //define voxels in surface
218.            for(i=0;i<SIZE_X;i++)
219.            {
220.               for(j=0;j<SIZE_Y;j++)
221.               {
222.                  for(k=0;k<SIZE_Z;k++)
223.                  {
224.                     /*if((j <SIZE_Y && VoxelWorldArray[i][j][k].exist == 1 &&
    VoxelWorldArray[i][j+1][k].exist == 0) ||
225.                        (j == 0 && VoxelWorldArray[i][j][k].exist == 1 ) ||
226.                        (i <SIZE_X && VoxelWorldArray[i][j][k].exist == 1 &&
    VoxelWorldArray[i+1][j][k].exist == 0) ||
227.                        (i >0 && VoxelWorldArray[i][j][k].exist == 1 && VoxelWorldArray[i-1][j]
    [k].exist == 0) ||
228.                        (k <SIZE_Z && VoxelWorldArray[i][j][k].exist == 1 &&
    VoxelWorldArray[i][j][k+1].exist == 0) ||
229.                        (j >0 && VoxelWorldArray[i][j][k].exist == 1 && VoxelWorldArray[i][j]
    [k-1].exist == 0))*/
230.                     if((j <SIZE_Y   && VoxelWorldArray[i][j][k].exist == 1  &&
    VoxelWorldArray[i][j+1][k].exist == 0) ||
231.                        (i <SIZE_X  && VoxelWorldArray[i][j][k].exist == 1  &&
    VoxelWorldArray[i+1][j][k].exist == 0) ||
232.                        (i >0       && VoxelWorldArray[i][j][k].exist == 1  &&
    VoxelWorldArray[i-1][j][k].exist == 0) ||
233.                        (k <SIZE_Z  && VoxelWorldArray[i][j][k].exist == 1  &&
    VoxelWorldArray[i][j][k+1].exist == 0) ||
234.                        (j >0       && VoxelWorldArray[i][j][k].exist == 1  &&
    VoxelWorldArray[i][j][k-1].exist == 0))
```

```
                {
                    VoxelWorldArray[i][j][k].isSurface= true;
                }
                else
                {
                    VoxelWorldArray[i][j][k].isSurface= false;
                }
            }
        }
    }
    initTriSurface();
}

void InitWorldV1(void)
{
    //float lightPos[]={50., 50., 10., 1.};
    int i,j,k;
    for(i=0;i<SIZE_X;i++)
    {
        for(j=0;j<SIZE_Y;j++)
        {
            for(k=0;k<SIZE_Z;k++)
            {
                VoxelWorldArray[i][j][k].exist= round(rand()/(float)RAND_MAX);
                //printf("%i ",VoxelWorldArray[i][j][k].Exist );
                //VoxelWorldArray[i][j][k].Density=0.5;
                //VoxelWorldArray[i][j][k].Color[4]=(1.0,1.0,1.0,1.0);
            }
        }
    }
    //surfaceOnly();
    //glEnable(GL_LIGHTING);
    //glEnable(GL_LIGHT0);
    //glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
}


void DrawSphere(float x, float y, float z)
{
    glTranslatef(x*SCALE,y*SCALE,z*SCALE);

    glScalef(SCALE, SCALE, SCALE);
    glPointSize(5.0);
    glBegin(GL_POINTS);
        glColor3f(0.0,0.5,0.0);
        glVertex3f( 0.5, 0.5,-0.5);
    glEnd();
    glScalef(1/SCALE, 1/SCALE, 1/SCALE);

    glTranslatef(-x*SCALE,-y*SCALE,-z*SCALE);
}
```

```
286.
287.
288.        //float color_numb = 1.0;
289.        void DrawCube(float x, float y, float z)
290.        {   //printf("%f %f %f \n", x,y,z);
291.            glTranslatef(x*SCALE,y*SCALE,z*SCALE);
292.
293.            //glColor3f(color_numb,color_numb,color_numb);
294.
295.            glScalef(SCALE, SCALE, SCALE);
296.            glBegin(GL_QUADS);      // Draw The Cube Using quads
297.                glColor3f(0.0,0.5,0.0);     //Green
298.                glVertex3f( 0.5, 0.5,-0.5);
299.                glVertex3f(-0.5, 0.5,-0.5);
300.                glVertex3f(-0.5, 0.5, 0.5);
301.                glVertex3f( 0.5, 0.5, 0.5);
302.                glColor3f(0.0,0.5,0.5);     //turquoise
303.                glVertex3f( 0.5,-0.5, 0.5);
304.                glVertex3f(-0.5,-0.5, 0.5);
305.                glVertex3f(-0.5,-0.5,-0.5);
306.                glVertex3f( 0.5,-0.5,-0.5);
307.                glColor3f(0.5,0.0,0.0);     //Red
308.                glVertex3f( 0.5, 0.5, 0.5);
309.                glVertex3f(-0.5, 0.5, 0.5);
310.                glVertex3f(-0.5,-0.5, 0.5);
311.                glVertex3f( 0.5,-0.5, 0.5);
312.                glColor3f(0.5,0.5,0.0);     //Yellow
313.                glVertex3f( 0.5,-0.5,-0.5);
314.                glVertex3f(-0.5,-0.5,-0.5);
315.                glVertex3f(-0.5, 0.5,-0.5);
316.                glVertex3f( 0.5, 0.5,-0.5);
317.                glColor3f(0.0,0.0,0.5);     //Blue
318.                glVertex3f(-0.5, 0.5, 0.5);
319.                glVertex3f(-0.5, 0.5,-0.5);
320.                glVertex3f(-0.5,-0.5,-0.5);
321.                glVertex3f(-0.5,-0.5, 0.5);
322.                glColor3f(0.5,0.0,0.5);     //Violet
323.                glVertex3f( 0.5, 0.5,-0.5);
324.                glVertex3f( 0.5, 0.5, 0.5);
325.                glVertex3f( 0.5,-0.5, 0.5);
326.                glVertex3f( 0.5,-0.5,-0.5);
327.            glEnd();            // End Drawing The Cube
328.            glScalef(1/SCALE, 1/SCALE, 1/SCALE);
329.
330.            glTranslatef(-x*SCALE,-y*SCALE,-z*SCALE);
331.        }
332.
333.        void drawVoxel()
334.        {
335.            int i,j,k;
336.            for(i=0;i<SIZE_X;i++)
```

```
337.            {
338.               for(j=0;j<SIZE_Y;j++)
339.               {
340.                  for(k=0;k<SIZE_Z;k++)
341.                  {
342.                     //printf("Drawing voxels");
343.                     //Out commenting isSurface so ALL existing Voxels are drawn.
344.                     if(/*VoxelWorldArray[i][j][k].isSurface &&*/ VoxelWorldArray[i][j][k].exist
    == 1)
345.                     {
346.                        DrawSphere((float)i, (float)j, (float)k);
347.                        DrawCube((float)i, (float)j, (float)k);
348.                     }
349.                  }
350.               }
351.            }
352.         }
353.
354.         void changeSize(int w, int h)
355.         {   // (you cant make a window of zero width).
356.            if (h == 0)
357.               h = 1;
358.
359.            float ratio =  w * 1.0 / h;
360.
361.            // Use the Projection Matrix
362.            glMatrixMode(GL_PROJECTION);
363.
364.            // Reset Matrix
365.            glLoadIdentity();
366.
367.            // Set the viewport to be the entire window
368.            glViewport(0, 0, w, h);
369.
370.            // Set the correct perspective.
371.            gluPerspective(45.0, ratio, 0.1, 100.0);
372.
373.            // Get Back to the Modelview
374.            glMatrixMode(GL_MODELVIEW);
375.         }
376.
377.         void cameraPos(void)
378.         {
379.            float sidestepX, sidestepY, sidestepZ;
380.
381.            if (pressedKeyUp == GLUT_KEY_UP)
382.            {
383.               camPosX+=camDirX*keySpeed;
384.               camPosY+=camDirY*keySpeed;
385.               camPosZ+=camDirZ*keySpeed;
386.            }
```

```
387.            if (pressedKeyDown == GLUT_KEY_DOWN)
388.            {
389.               camPosX-=camDirX*keySpeed;
390.               camPosY-=camDirY*keySpeed;
391.               camPosZ-=camDirZ*keySpeed;
392.            }
393.            if (pressedKeyLeft == GLUT_KEY_LEFT)
394.            {
395.               //(a2*b2 - a3*b2)
396.               sidestepX = calcCrossProdX(camUpY,camDirZ, camUpZ, camDirY);
397.               //(a3*b1 - a1*b3)
398.               sidestepY = calcCrossProdY(camUpZ, camDirX, camUpX, camDirZ);
399.               //(a1*b2-a2*b1)
400.               sidestepZ = calcCrossProdZ(camUpX, camDirY, camUpY, camDirX);
401.               camPosX+=sidestepX*keySpeed;
402.               camPosY+=sidestepY*keySpeed;
403.               camPosZ+=sidestepZ*keySpeed;
404.            }
405.            if (pressedKeyRight == GLUT_KEY_RIGHT)
406.            {
407.               //(a2*b2 - a3*b2)
408.               sidestepX = calcCrossProdX(1,camDirZ, 0, camDirY);
409.               //(a3*b1 - a1*b3)
410.               sidestepY = calcCrossProdY(0, camDirX, 0, camDirZ);
411.               //(a1*b2-a2*b1)
412.               sidestepZ = calcCrossProdZ(0, camDirY, 1, camDirX);
413.               camPosX-=sidestepX*keySpeed;
414.               camPosY-=sidestepY*keySpeed;
415.               camPosZ-=sidestepZ*keySpeed;
416.            }
417.         }
418.         void drawAimsight()
419.         {
420.            glColor3f(0.0,1.0,0.0);
421.            glPointSize(10.0);
422.            glBegin(GL_POINTS);
423.               glVertex3f(camPosX+camDirX,camPosY+camDirY,camPosZ+camDirZ);
424.            glEnd();
425.         }
426.
427.         void drawTriSurface()
428.         {
429.            for(int i=0; i<triangleAmount; i++)
430.            {
431.               glBegin(GL_TRIANGLES);      // start draw triangle
432.                  glColor3fv(TriangleSurfaceArray[i].color);      //Green
433.                  glVertex3fv(TriangleSurfaceArray[i].cornerA);
434.                  glVertex3fv(TriangleSurfaceArray[i].cornerB);
435.                  glVertex3fv(TriangleSurfaceArray[i].cornerC);
436.               glEnd();           // End Drawing triangle
437.            }
```

```
438.        }
439.
440.        void renderScene(void)
441.        {   // Camera Movement
442.            cameraPos();
443.
444.            // Clear Color and Depth Buffers
445.            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
446.
447.            // Reset transformations
448.            glLoadIdentity();
449.            // Set the camera
450.            //gluLookAt(10.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
451.            //gluLookAt(x, y, z,x+lx, y+ly, z+lz, 0.0, 1.0, 0.0);
452.            gluLookAt(  camPosX,        camPosY,        camPosZ,
453.                        camPosX+camDirX,camPosY+camDirY, camPosZ+camDirZ,
454.                        camUpX,         camUpY,          camUpZ);
455.
456.            // Animate voxels - rotations around the objects
457.            //glRotatef(angle, 0.0, 1.0, 0.0);
458.
459.            // Draw voxels and Voxels starting points
460.            drawVoxel();
461.
462.            drawAimsight();
463.
464.            //Draw triangle surface
465.            //drawTriSurface();
466.
467.            glutSwapBuffers();
468.        }
469.
470.        void pressKey(int key, int xx, int yy)
471.        {
472.            switch (key)
473.            {
474.                case GLUT_KEY_UP :
475.                    pressedKeyUp = key;
476.                    break;
477.                case GLUT_KEY_DOWN :
478.                    pressedKeyDown = key;
479.                    break;
480.                case GLUT_KEY_LEFT :
481.                    pressedKeyLeft = key;
482.                    break;
483.                case GLUT_KEY_RIGHT :
484.                    pressedKeyRight = key;
485.                    break;
486.            }
487.        }
488.
```

```
489.        void releaseKey(int key, int x, int y)
490.        {
491.           switch (key)
492.           {
493.              case GLUT_KEY_UP :
494.                 pressedKeyUp = 0;
495.                 break;
496.              case GLUT_KEY_DOWN :
497.                 pressedKeyDown = 0;
498.                 break;
499.              case GLUT_KEY_LEFT :
500.                 pressedKeyLeft = 0;
501.                 break;
502.              case GLUT_KEY_RIGHT :
503.                 pressedKeyRight = 0;
504.                 break;
505.           }
506.        }
507.
508.        void mouseMove(int x, int y)
509.        {
510.           //printf("%i %i \n", x,y);
511.           mouseDeltaX = (x - mouseCenterX)* mouseSpeed;
512.           mouseDeltaY = (y - mouseCenterY)* mouseSpeed;
513.
514.           // update camera's direction
515.           camDirX = -sin(mouseDeltaX);
516.           camDirZ = cos(mouseDeltaX);
517.           camDirY = -sin(mouseDeltaY);
518.           camDirZ +=  cos(mouseDeltaY);
519.
520.           float lenght = sqrt(camDirX*camDirX + camDirY*camDirY +
    camDirZ*camDirZ);
521.            camDirX /= lenght;
522.            camDirY /= lenght;
523.            camDirZ /= lenght;
524.            //lenght = sqrt(camDirX*camDirX + camDirY*camDirY + camDirZ*camDirZ);
525.            //printf("%f \n", lenght);
526.        }
527.
528.        void mouseButton(int button, int state, int x, int y) {
529.
530.           // only start motion if the left button is pressed
531.           if (button == GLUT_LEFT_BUTTON)
532.           {
533.              // when the button is released
534.              if (state == GLUT_UP)
535.              {
536.                 checkHitVoxel(camPosX+camDirX, camPosY+camDirY,
    camPosZ+camDirZ);
537.              }
```

```
538.                else
539.                {// state = GLUT_DOWN
540.                }
541.             }
542.          }
543.
544.          int main(int argc, char **argv)
545.          {   //Init GLUT and create window
546.             glutInit(&argc, argv);
547.             glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
548.             glutInitWindowPosition(1080,0);
549.             glutInitWindowSize(windowSizeX,windowSizeY);
550.             glutCreateWindow("Voxels");
551.             glEnable(GL_DEPTH_TEST);
552.
553.             InitWorld();
554.
555.             //Register callbacks
556.             glutDisplayFunc(renderScene);
557.             glutReshapeFunc(changeSize);
558.             glutIdleFunc(renderScene);
559.
560.             //Keyboard calls
561.             glutIgnoreKeyRepeat(1);
562.             glutSpecialFunc(pressKey);
563.             glutSpecialUpFunc(releaseKey);
564.
565.             //Mouse calls
566.             glutMouseFunc(mouseButton);
567.             //glutMotionFunc(mouseMove);
568.             glutPassiveMotionFunc(mouseMove);
569.
570.             //Enter GLUT event processing cycle
571.             glutMainLoop();
572.
573.             return 1;
574.          }
```

VoxelStruct.h

```
1.  typedef struct Voxel
2.  {
3.      int exist;
4.      float density;
5.      float color[4];
6.      bool isSurface;
7.  } Voxel;
8.
9.  typedef struct Triangle
10. {
11.     float color[4];
```

```
12.     float cornerA[3];
13.     float cornerB[3];
14.     float cornerC[3];
15.     float normal[3];
16. } Triangle;
```