Developing a Comprehensive Software Toolkit for Creating Digital Mosaic Artwork

Dmitry Bosikov
CUNY City College

How does access to this work benefit you? Let us know!
Follow this and additional works at: http://academicworks.cuny.edu/cc_etds_theses
Part of the Computer Sciences Commons

Recommended Citation
http://academicworks.cuny.edu/cc_etds_theses/87

This Thesis is brought to you for free and open access by the City College of New York at CUNY Academic Works. It has been accepted for inclusion in Master's Theses by an authorized administrator of CUNY Academic Works. For more information, please contact AcademicWorks@cuny.edu.
DEVELOPING A COMPREHENSIVE SOFTWARE TOOLKIT FOR RENDERING DIGITAL MOSAIC ARTWORK

THESIS
Submitted in partial fulfillment of
the requirement for the degree

Master of Computer Science
at
The City College
of the
City University of New York

by
Dmitry Bosikov
September 2012

Approved:

______________________________________________
Professor George Wolberg, Thesis Advisor

Department of Computer Science
Table of Contents

Abstract ...........................................................................................................3

1 Getting started with GLSL .................................................................4
  1.1 Introduction GLSL.......................................................................4
  1.2 Pipeline.......................................................................................5
  1.3 Vertex and Fragment Shader......................................................6

2 Creating Basic functionality with GLSL...........................................10
  2.1 Shaders creating.........................................................................10
  2.2 Working with shader.................................................................13
  2.3 Vertices manipulation functions...............................................20

3 GLSL Mosaic........................................................................................26
  3.1 Introduction to mosaic creation.................................................26
  3.2 Mosaic initialization.................................................................27
  3.3 Draw mosaic..............................................................................34
  3.4 Light & camera..........................................................................37

Conclusion..............................................................................................40

Bibliography..........................................................................................41

Appendix A. Rendering Results............................................................43

Appendix B. Program code.................................................................50
Abstract

This project is based on the OpenGL Shading Language (GLSL) that facilitates real time control of 3D rendering properties such as viewing and lighting. GLSL also accelerate rendering speed and reduces the time for creating and testing 3D models. The goal of this project is to create a 3D software package that renders a realistic mosaic in real time. To achieve this goal we exploit GLSL and demonstrate its use in this paper.

A realistic mosaic is treated as a collection of 3D tiles that interact with a light source and a viewing camera. The simplest example of interaction is a moving light source around the mosaic that makes its appear lighter or darker. The main objective of the proposed software is to visualize mosaics in areas such as model building, movies, games, and design.

The project includes three steps: learning GLSL, using it to visualize basic 3D objects, generating renderings of realistic mosaics. As result of this work, designed software works at real time rates.
1 Getting started with GLSL

1.1 Introduction GLSL

OpenGL is a software interface to graphics hardware. The interface consists of a set of several hundred functions that allow the programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects. It is not just a set of commands for drawing 3D models, but it also controls rendering properties and creates sophisticated rendering. These functions allow the creation of high complexity programs that work with graphics, such as 3D editors and games. However, the technology continues to move forward, and as result, we nowadays have GLSL.

OpenGL Shading Language (GLSL) is a high-level shading language based on the syntax of the C programming language. GLSL is an extension of OpenGL. It increases functionality in many times compared to the previous version. To be more specific, it gives an ability to create your own functionality inside a GPU. In OpenGL, information is sent to a GPU which responds to requests and returns results. In GLSL, you have the ability to change the process that happens inside the hardware. You can create your own algorithms for managing 3D objects in a graphic scene: drawing objects
on a screen, substructuring or changing them on the fly depends on the camera and object location. GPU memory is also opened for access that allows the saving of all object data inside a GPU, furthermore reducing the time for transferring data to a GPU, so we can spend more time on rendering an efficient and realistic graphic scene.

1.2 Pipeline

Pipeline is an array of execution blocks that operates with data that are sent for rendering. Currently, there are six shaders which construct the pipeline. In this work, we are use only two of them: vertices and the fragment shader. They are written in GLSL, which is similar to the C language, and are two main shaders are implemented in the program. GLSL uses the same syntax as the C language, but it has extra functions, structures, and classes for graphic purposes. For example, GLSL has classes: vec4, vec3, vec2, mat4, mat3, and mat2 which represent vectors and matrixes. Inside this language, we can also find many functions used in 3D graphics, such as the dot product or matrix normalization that makes writing software much easier and more enjoyable.
1.3 Vertex and Fragment Shader

A vertex shader is responsible for the manipulation of vertices. It is usually used for object transformation, light calculation, and camera projection. For example, we can transform an object using rotation, scale and translate. After an object is placed in a calculated location, lighting effects must be applied. The simplest effect can be calculated by using vectors of vertices, camera, and light positions. Lighting computation is very complex, especially while working with shaders. The last important object of the graphic scene is a camera. This object does not physically exist, but it is just an extra transformation that continues moving and scaling objects that are creating an illusion of a camera. For example, to rotate a camera to the right, it is necessary to see the objects located on the right side. All objects are taken and rotated in opposite directions around the camera as the objects end up in front of the camera.

After a vertex shader finishes manipulating data, it sends it to a fragment shader that moves the data from the vertex to pixels and draws results on a screen. During the simplest transition from vertices to pixels, pixels are filled with color inside the edges that connect the vertices together. We can put color or map texture between the edges that join the
vertices. This process is not as simple as it seems at first glance. First of all, there are different ways to fill a space with color. You can take one average color from several vertices or use a color gradient between the vertices. A simple example is to put a piece of texture in the space between the edges. However, if we want to join two or more textures together, for example, you have the skin and clothes texture of a person, it is easy to put skin texture on the lowest level, put the clothes on the next level, and just have a part of clothes become invisible. We use this method because we want to animate the movement of the clothes that moves by wind. To achieve it, we have to make random changes of the invisible layer of the clothes.

Front and back faces are also good example of filling in space between edges with color. Depending on the location of the camera, a viewer can see the front or back face. Each face can have different lighting and textures.

As you can see, there are many different things that need to be counted for creating the realistic graphic scene. Shaders are a powerful tool that allows one to create what one wants, remove restrictions, and give freedom for new achievements.

The main() function connects shaders together using the OpenGL library. Two types of variables are used in this process. The first represents
the variables’ IDs inside the shaders and the second stores values. To set up and use shaders, we need to create shader code that must be compiled and linked on GPU. After the success of these operations, a resulting shader program is ready to work. In the main() function, we are going to create buffers and variables for shaders. Each buffer or variable in a shader will have an assigned ID in main() function that can be stored in an array of variables. These IDs are needed for sending data to a GPU. IDs let the shader know what variables should be received a value or structure. After all IDs are assigned and the buffers are set, the data can be sent to the shaders. When the shader program concludes, we need to free up all the resources that were used. Another feature is ability to manipulate the resources to allow us to have multiple versions of shaders and buffers.
Summary

It is important to remember that OpenGL is the technology that allows the drawing of 3D objects with a GPU. Recently, OpenGL moved to a programmable pipeline that allows the creation of functionality which works with vertices and can be drawn on a screen. There are two main shaders that must be created for running a program. They are the vertex and fragment shaders. All use GLSL shader language that is similar to C language and includes a variety of functions and constants for operations over matrixes and vertices. Finally, before using shaders, they need to be compiled and linked into a program.
2  Creating Basic functionality with GLSL

2.1 Shaders creating

This chapter reviews the code of a simple GLSL program that is used in this project. We begin by creating a shader. In each shader, we need to create the following functions: defining the version of the shader, creating uniform variables, “in” and “out” arrays of data, and the main function. If we follow the order, we first have to define a version. The version is a very important component due to the fast development of GLSL technology as not all GPUs support the latest version of GLSL. A version of language must be set as a number in the first line of a shader. Sometimes, languages used in sending data between shaders in different versions can be different. Below is an example of defining a version:

```plaintext
#version 400
```

Code 2.1.- Defining shader version

The version used in this project needs to be defined on the top of the code. If a version is not defined, the GLSL’s compiler will select the first stable version of GLSL, so as you can understand, the code for each version
will be different. In this example, we will use the version 400. After defining the version, we are moving to variables and arrays. To define variables, we need to write the next code:

```cpp
uniform mat4 ModelMatrix;
uniform vec4 LightLocation;
uniform float SpecularExp;
```

Code 2.2. – Defining shader variables

To define a variable, you need to type both “uniform” and variable’s type. Uniforms are program variables that are constant during program execution. Mat4 is a matrix four by four and vec4 is a vector(x, y, z, w coordinates). Values for these variables will be set up in the main() function and can be reset at any time between execution of the shader program. The data that will be sent from the main() function is:

```cpp
in vec4 vertex;
in vec4 vertexColor;
in vec4 vertexNormal;
out vec4 pointColor;
```

Code 2.3. – Defining shader arrays

In the above code we define buffers for vertex coordinates, colors, and the normal from the vertices. Buffers are set as “in”; it means that the data
will be sent from the program to the shaders. The last variable that starts
with “out” is a result of a calculation of the vertex shader and the data will
be then sent to the next shader. In our case, it is a fragment shader.

The next step is the creation of the main() function, which as in C
language, serves as an entry point into the execution of the shader, so you
can create many functions in the shaders, but they all will be called from the
main() function. Here is an example:

```c
void main(void){
    gl_Position = vertex;
    pointColor = vertexColor;
}
```

Code 2.4. – Shader entry function

As you can see from the program above, it is very simple code. It does
set up a gl_Position, which is a vertex position. This position in future
examples will be multiplied by the transformation matrixes that were
mentioned in Chapter 1. The next line is sending the vertexColor to a
fragment shader. In this case, the fragment shader will also be very simple. It
just uses color that was sent from the vertex shader and is set up for
rendering on a screen.

```c
#version 400
in vec4 pointColor;
```
out vec4 out_color;
void main(void){
    out_color = pointColor;
}

Code 2.5.- Simple fragment shader

2.2 Working with shader

The first step in working with shaders is initialization. This process consists of compiling, linking and starting the shader program. We are going to review step by step the creation of this program. First, we will read a shader code. We are going to use the ReadShader function that will read a shader file contained in the “vertexShaderFile” variable. Initially, a shader can be created in a text string variable, but this case is hard to manage in a code. We create a text variable for storing and saving a shader code in it.

const GLchar* shaderSource;
shaderSource = ReadShader(vertexShaderFile);

Code 2.6.- Reading shader code

When we have a shader code in a text variable, we need to upload the shader to the GPU memory and compile it. In process of compiling, we must set a type of the shader that is currently in use. The three next parameters
will define source that hold shader code: number of elements in the string,
an array of pointers to strings containing the shader code and an array of
string lengths.

VertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(VertexShaderId,1,&shaderSource,NULL);

Code 2.7.- Shader creation

These operations need to be repeated for a fragment shader. Then two
compilations must be attached and linked together. In order to make a shader
program, we have to create a new program (glCreateProgram) and its ID
using linked results.

ShaderProgramID = glCreateProgram();
glAttachShader(ShaderProgramID, VertexShaderId);
glAttachShader(ShaderProgramID, FragmentShaderId);
glLinkProgram(ShaderProgramID);

Code 2.8.- Shader program creation

The last step is to run a shader program.

glUseProgram(ShaderProgramID);

Code 2.9.- Running shader program
When the shader program is started, we have to connect the variables of the shaders with the variables of the main() function. There are two main types, arrays and uniforms, that we will work with. Sending uniform values is very simple, so we can start with it. The first thing that needs to be done is getting the IDs of the shader variables that we want to send data for. For example, we want to send a model matrix to the shader. In the vertex shader, this uniform variable is called “ModelMatrix”, so to get the ID of this uniform variable, we need to write the next lines:

```c
unsigned int modelMatrixId;
modelMatrixId=glGetUniformLocation(ShaderProgram, “ModelMatrix”);
```

Code 2.10.- Getting shader variable ID

It is obvious that we create variables to store uniform variables’ ID of shaders and we get these IDs by calling a glGetUniformLocation. ShaderProgram is an ID of the shader program which we compiled and linked in the previous section. Now when we have the ID of the variable which we can transfer to a shader:

```c
glUniformMatrix4fv(modelMatrixID,1,false,modelMatrix.m);
```

Code 2.11.- Sending variable to shader
The `glUniformMatrix` is a function that allows us to send the data to a shader. If you used OpenGL before, you already know that OpenGL is written as structure (not OOP) and for each type of variable, you have your own function. Thus, if you want to send an integer, you have to write a `glUniformInt(..)`. In our case, we have the four-by-four matrix, float elements, and the `v` represents pointer. We send a variable ID to a function, the number of elements, do the conversion to float if needed, and send the value of variable.

Now, we know how to send variables to a shader. Before sending the array, we need to create an object that will define what is in an array (vertex array object). The next step is the creation of an array (buffer array object), and the last is sending the data to the shader. We are going to create two variables that will hold vao and bao IDs.

```c
unsigned int vao, bao;
```

**Code 2.12.- Shader object IDs**

First, we create a variable to store vao ID. After that we create vao ID, bind it with the shader, which makes all changes apply to an object. If you rebind another object, the changes will be applied to the current one. If you
would like to use an array, you need to bind the vao before using. The
definition of vao is:

```c
glGenVertexArrays(1,&vao);
glBindVertexArray(vao);
```

Code 2.13.- Creating object in shader

In the first line, we create a vao. The first parameter in this function is
a count of how many objects you want to create, and the second is an
address of the variable. In the second line we bind the vao that makes it
active. The next step is creating of an array:

```c
glGenBuffers(1,&bao);
glBindBuffer(GL_ARRAY_BUFFER,bao);
glBufferData(GL_ARRAY_BUFFER, sizeofVertex, &vertexs[0],GL_STATIC_DRAW);
```

Code 2.14.- Creating buffer in shader

As in the case with the vao, the first command creates the buffer
objects and saves the IDs of the bao variables there. The next line sets the
buffer as an active buffer. And the last line defines the buffer and sends data
to the GPU memory. It tells us that the memory which we reserved is a
GL_ARRAY_BUFFER. The sizeOfVertex is the size of an array we want to
define; &vertexs[0] is a pointer on the data of an array; the last parameter
defines the location of the buffer in the GPU memory. If the data is changed
continuously, we will need to put the buffer in DYNAMIC memory, and if the
data is not changed, we will put it in the STATIC. In our case, we use the
GL_STATIC_DRAW. The next step is connecting the shader variables with
the buffer. For that, we need to define pointers that will tell the GLSL what
data variables in the main() function should be used. In the code below we
will define that our buffer will have structures created from vertices
coordinates and vertices colors:

```c
unsigned int vertex = glGetAttribLocation(ProgramId, “vertex”);
glEnableVertexAttribArray(vertex);
unsigned int vertexColor = glGetAttribLocation(ProgramId, “vertexColor”);
glEnableVertexAttribArray(vertexColor);
glVertexAttribPointer(vertex, 4, GL_FLOAT, GL_FALSE, sizeof(vertex[0]),
BUFFER_OFFSET(0));
glVertexAttribPointer(vertexColor, 4, GL_FLOAT, GL_FALSE, sizeof(vertex[0]),
BUFFER_OFFSET(sieofof(vertex[0].vectors)));
```

Code 2.15.- Enable shader arrays

In the first four lines of this declaration, we create variables that store
the IDs of buffer variables. Next, we enable buffer variables in the shader to
send data to them. In the last two lines, we define a pointer for buffer
variables. The first parameter is an ID of a buffer variable in the shader, the second is a count of the sent values, the third is a type of sent values, the fourth does conversion to float if needed, the fifth is the size of the vertex object, and the last is an offset of values to start, which represents the coordinates in the first line and the colors in the second.

Now that we have defined and set up all the needed information, we can draw some objects. We are going to use glBufferData with &vertex[0]. A vertex variable has coordinates and colors inside. To draw these vertices on a screen, we do the following:

```gl
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Code 2.16.- Draw an object

In the function above, we specified that we will draw triangles, use the data from the first element, and take three elements. As result, we will have a drawn triangle on a screen.
2.3 Vertices manipulation functions

After we created the shaders and connected them to the main() function, we need to write a few common functions that manipulate the arrays of vertices with matrixes. These are rotation, translation, and projection. We will not stop on a deep explanation of each function. A code will be introduced, that will allow us to start using functions quickly. We are going to discuss the functions that manipulate with the matrixes because the understanding of them is very important in the future working with shaders. Now, we will present the math functions which are used in a vector calculation:

```c
void CrossProduct3(GLfloat *ret, GLfloat *a, GLfloat *b) {
    ret[0] = a[1]*b[2] - a[2]*b[1];
    ret[1] = a[2]*b[0] - a[0]*b[2];
    ret[2] = a[0]*b[1] - a[1]*b[0];
}
GLfloat glmosaic::DotProduct3(GLfloat* a, GLfloat* b) {
    GLfloat ret = 0;
    int i;
    for (i = 0; i < 3; i++) {
        ret += a[i]*b[i];
    }
    return ret;
}
GLfloat glmosaic::Magnitude3(GLfloat *a) {
    return sqrt((a[0]*a[0]) + (a[1]*a[1]) + (a[2]*a[2]));
}
void glmosaic::NormalizeVector3(GLfloat *a) {
    GLfloat magnitude;
```
GLint i;
magnitude=Magnitude3(a);
for(i=0;i<3;i++)
    a[i]=a[i]/magnitude;
}

GLfloat glmosaic::RadiansToDegrees(float radians){
    return radians*(GLfloat)(180/PI);
}

Matrix glmosaic::MultMatrix(Matrix *m1, Matrix *m2){
    Matrix ret;
    int i,j,k;
    for(i=0;i<4;i++){
        for(j=0;j<4;j++){
            ret.m[gli(i,j)]=0.0;
            for(k=0;k<4;k++)
                ret.m[gli(i,j)]+=m1->m[gli(i,k)]*m2->m[gli(k,j)];
        }
    }
    return ret;
}

Code 2.17.- Common use functions

These functions were mentioned before all manipulations with the vertices were done through matrices. We look on a vertex as the sum of three vectors x, y, z. It is proper to use a matrix in calculation since in the matrix, you can apply any vertex transformation: scaling, rotation, and translation.

Try to understand how this transformation is done. For example, we have a matrix with a transformation “M” and a vertex “V” which we want to transform.
M multiplied by V does the trick. Next, we will look at what happened inside this multiplication:

\[
\begin{pmatrix}
a_1 & a_2 & a_3 & a_4 \\
b_1 & b_2 & b_3 & b_4 \\
c_1 & c_2 & c_3 & c_4 \\
d_1 & d_2 & d_3 & d_4
\end{pmatrix}
\times
\begin{pmatrix}
x \\
y \\
z \\
w
\end{pmatrix}
=
\begin{pmatrix}
a_1 \cdot x + a_2 \cdot y + a_3 \cdot z + a_4 \cdot w \\
b_1 \cdot x + b_2 \cdot y + b_3 \cdot z + b_4 \cdot w \\
c_1 \cdot x + c_2 \cdot y + c_3 \cdot z + c_4 \cdot w \\
d_1 \cdot x + d_2 \cdot y + d_3 \cdot z + d_4 \cdot w
\end{pmatrix}
\]

You can see that after the multiplication on the matrix, it causes the dependence of the coordinates on each other. In this case, if z-coordinate decreases, x and y coordinates will change, and it has the effect of the object moving away from the camera.

There are some standard transformations which use matrixes. The examples are shown below:

Matrix glmosaic::ProjectionMatrix(float fovy, float aspect_ratio, float near_plane, float far_plane) {
    Matrix ret = { { 0 } };
    const float y_scale = Cotangent(DegreesToRadians(fovy / 2)),
                x_scale = y_scale / aspect_ratio,
                frustum_length = far_plane - near_plane;
    ret.m[0] = x_scale;
    ret.m[5] = y_scale;
    ret.m[10] = -((far_plane + near_plane) / frustum_length);
    ret.m[11] = -1;
    ret.m[14] = -((2 * near_plane * far_plane) / frustum_length);
    return ret;
}
This function sets up the matrix so than an object can be seen in perspective view. The following function makes the orthographic view:

```cpp
Matrix glmosaic::OrthoMatrix(GLfloat x2, GLfloat x1, GLfloat y2, GLfloat y1, GLfloat z2, GLfloat z1){
    Matrix ret ={{0}};
    ret.m[0]=2/(x1-x2);
    ret.m[5]=2/(y1-y2);
    ret.m[10]=-2/(z1-z2);
    ret.m[14]=-(-z1+z2)/(z1-z2);
    ret.m[15]=1.0;
    return ret;
}
```

Code 2.20.- Orthogonal matrix

As you can see, in each function we set dependences between the vector coordinates. In the same way, we can create basic transformations such as rotation, scale and translation:

```cpp
void glmosaic::Rotate(Matrix *m1, GLfloat angle, GLfloat x, GLfloat y, GLfloat z){
    angle = DegreesToRadians(angle);
    Matrix xMatrix =
    {{1.0, 0.0, 0.0, 0.0,
      0.0, cos(angle), -sin(angle), 0.0,
      0.0, sin(angle), cos(angle), 0.0,
      0.0, 0.0, 0.0, 1.0
    };
    Matrix yMatrix={{
        {cos(angle), 0.0, sin(angle), 0.0,
        0.0, 1.0, 0.0, 0.0,
        -sin(angle), 0.0, cos(angle), 0.0,
        0.0, 0.0, 0.0, 1.0
    };
```
Matrix zMatrix={
{cos(angle),-sin(angle), 0.0, 0.0,
sin(angle),cos(angle), 0.0, 0.0,
0.0, 0.0, 1.0, 0.0,
0.0, 0.0, 0.0, 1.0
};
Matrix ret;
Identity(&ret);
if(x==1.0)
    ret = MultMatrix(&xMatrix,&ret);
if(y==1.0)
    ret = MultMatrix(&yMatrix,&ret);
if(z==1.0)
    ret = MultMatrix(&zMatrix,&ret);
*m1= MultMatrix(&ret,m1);
}
void glmosaic::Scale(Matrix *m1, GLfloat x, GLfloat y, GLfloat z){
    Matrix ret;
    Identity(&ret);
    ret.m[0]=x;
    ret.m[5]=y;
    ret.m[10]=z;
    *m1=MultMatrix(&ret,m1);
}
void glmosaic::Translate(Matrix *m1, GLfloat x, GLfloat y, GLfloat z){
    Matrix ret;
    Identity(&ret);
    ret.m[12]=x;
    ret.m[13]=y;
    ret.m[14]=z;
    //PrintMatrix(&ret);
    *m1=MultMatrix(&ret,m1);
}

Code 2.21.- Object transformation functions
Summary

In this part we looked through the configuration and the use of shaders. They are: creating the shader program, setting up shaders’ components, and sending commands for drawing. It is important to make these steps clear and understand each function used. One small mistake and you will see only a black screen. It is very hard to debug drawing errors since despite the fact that this program works correctly, it can make your objects go too far from the camera, or put them behind the camera, or make the same color as the background. While using the drawing library, all steps need to be clear and well documented. Now, we know how to manipulate a vertex with matrixes that allow use to rotate, the scale, and translate the objects which are created by vertices.
3 GLSL Mosaic

3.1 Introduction to mosaic creation

This chapter is concerned with the mosaic creation. In the first part, we will discuss a theoretical part. As it was mentioned before, the goal of this project is to create a realistic mosaic using minimum resources. To minimize resources, we decided to use two graphics figures – bump and texture mapping. Texture mapping is filling in a mosaic with images of tiles. Bump mapping involves a little more complex technology. On a simple example, we can see that bump mapping puts black spots on an image and make an illusion of 3D. In GLSL, bumps will be created not with black spots, but with updating normals from vertices. We use a vertex normal because it is the main part of the light calculation. Light is one of the main concerns in this project.

As to mosaic creation, the way to reduce using resources in 3D graphics by the simplifying the object geometry. Therefore, we create the mosaic as a flat plane with multiple vertices inside. In a mosaic, we need at list four vertices for each tile. Because this is a test project and we do not know how many vertices are enough to generate a realistic mosaic, we add a parameter which allows us to change the number of vertices per a tile. Now,
that we know that mosaic will be array of vertices, we omit use of z coordinate, we assume that tiles are flat.

The next step is to choose a structure that will have our vertices. It is understandable that it will use the coordinate system. We will also need a color value for calculation of light effects on the vertex, normal coordinates for bump mapping, and texture coordinates for texture mapping. As result, our vertex is a structure with coordinates, color, normal and texture coordinates.

3.2 Mosaic Initialization

This chapter will go through the initialization of the mosaic function. We will look in detail through the functions that will generate the mosaic vertices and send an array to GPU memory. This process includes two parts, the first of them is the generation of the mosaic which stored in a buffer, the second is initialization of GLSL variables and the sending of the generated information to the GPU memory. Generation of the matrix function is:

```cpp
Vertex* UpdateMosaicVertexNormals(){
    float d; int r;
    Vertex *vertices;
    Vertexs = new Vertex[nvertices];
    ```
int hstrip, stripx2, i, j, ij;
unsigned int xc=0, yc=0;
float xgrid=0, ygrid=0;

Code 3.1.- Storage creation for mosaic

In the first line, we define the function that creates a vertex array and sends it back. In the second line, we define two variables that we will use for storing random numbers for bump mapping. In the next two lines, we define a vertex array that will hold generated information. Next, we initialize variables with a number of vertices in rows and columns. Variables xc and yc will give the ability to find borders between tiles. The last line stores the offset between the coordinates for the next vertex.

In the next step, we are going to enter the loop which goes through all the vertices in the mosaic:

for(i=0;i<stripsx2;i++){
    for(j=0;j<hstrip;j++){
        ij=(i*hstrip)+j;
    }
}

Code 3.2.- Mosaic generation loop

This code is straight forward with the exception of the variable “ij”. This is needed because in the GLSL matrix, access is not represented
horizontally but vertically. In this variable, we just reversed the i and j indices. We can see how coordinates are done for mosaic:

```cpp
vertexs[ij].vectors[0]=xgrid;
vertexs[ij].vectors[1]=ygrid;
vertexs[ij].vectors[2]=0.0;
vertexs[ij].vectors[3]=1.0;
if( (xc<mborder) | (xc==mborder+mquad-1) ){
    xgrid+=bmesh;
} else{
    xgrid+=mesh;
}
} //exit from j loop;
if( (yc<mborder) | (yc==mborder+mquad-1) ){
    ygrid+=bmesh;
} else{
    ygrid+=mesh;
}
Xgrid=0.0;

Code 3.3.- Mosaic coordinates

In this code, we update offset in the coordinate system for each vertex and save these values to the active vertex. We use “if” statements for checking if we are on a border or on a tile. The color creation for the mosaic is:

```cpp
yc=(i)%(mborder+mquad);
xc=(j)%(mborder+mquad);
if( (xc<mborder) | (yc<mborder)){
    vertexs[ij].colors[0]=0.0;
```
vertices[ij].colors[1]=0.0;
vertices [ij].colors[2]=0.0;
vertices [ij].colors[3]=1.0;
}
else{
    vertices [ij].colors[0]=materialColorlvl;
    vertices [ij].colors[1]=materialColorlvl;
    vertices [ij].colors[2]=materialColorlvl;
    vertices [ij].colors[3]=materialColorlvl;
}

Code 3.4.- Mosaic colors

If the vertex is located on a border or tile, we check in the same way and assign a color for it. In the program we put a black color for the border and the defined colors for the tile. We also calculate yc and xc values that will tell us whether we are on a border or not. We add the sum of the vertices in the border and the tile, and then divide the number of the current vertex of this sum and save the remainder. We are getting close to the finish of this function and continue with normal calculation:

vertices[ij].normal[0]=0.0;
vertices [ij].normal[1]=0.0;
vertices [ij].normal[2]=1.0;
vertices [ij].normal[3]=0.0;
if(updateNormals>0){
    if(xc>=mborder){
        if(yc==mborder){
            RotateVector(&vertices [ij],-normalAngle,1.0,0.0,0.0,0.0);
        } else if(yc==mborder+mquad-1)
            RotateVector(&vertices [ij],normalAngle,1.0,0.0,0.0,0.0);
    }
if(yc>=mborder){
  if(xc==mborder){
    RotateVector(&vertices [ij],normalAngle,0.0,1.0,0.0);
  }
  else if(xc==mborder+mquad-1){
    RotateVector(&vertices [ij],-normalAngle,0.0,1.0,0.0);
  }
}

d = (float)GenRandNumber(maxAngle);
r = GenRandNumber(maxR);
if(d>0){
  if(((xc>mborder)&&(yc>mborder)) && ((yc>=mborder)&&(xc>mborder)))
    switch(r){
      case 1:
        RotateVector(&vertices [ij],-d,1.0,0.0,0.0);
        break;
      case 2:
        RotateVector(&vertices [ij],d,1.0,0.0,0.0);
        break;
      case 3:
        RotateVector(&vertices [ij],-d,0.0,1.0,0.0);
        break;
      case 4:
        RotateVector(&vertices [ij],d,0.0,1.0,0.0);
        break;
      default:
        break;
    }
}

Code 3.5.- Mosaic bump mapping

This calculation includes three parts. In the first part, we define a normal with a default value for each vertex. Since the surface is flat, we do not need to calculate the normal for each vertex, therefore saving great amount of time. Next, we check if we need to update normals along the edges of the mosaic. If not, we skip this step. Otherwise, we check whether
the vertex is located on the edge of the tile, and then the normal rotates by
the angle defined in the program. Finally, the last part is working with the
normal vectors. We generate random numbers and use them as controls for
updating the normals. Then we update in the same way as before by rotation,
but now we use a random angle.

In the last part of the creating the mosaic, we have to set the texture
coordinates for each vertex. This part is simple. We know how many
vertices are in one tile and we know a size of the texture. All textures have a
ranging size from 0 to 1. We simply multiply number of vertices in the tile
by the distance between the vertices, and we check if the vertex is located on
the border or the tile.

if(!xc<mborder)
    vertices [ij].texel[0]=(xc-mborder)*(1.0/mquad);
else
    vertices [ij].texel[0]=0;
if(!yc<mborder)
    vertexs[ij].texel[1]=(yc-mborder)*(1.0/mquad);
else
    vertices [ij].texel[1]=0.0;
vertices [ij].texel[2]=0.0;
vertices [ij].texel[3]=0.0;

Code 3.6.- Mosaic texture coordinates

After we finish generating the mosaic vertices, we can transfer them
to a GUP memory. As it was done in the second chapter, we need to create
vao and bao elements, activate buffer variables in the shader, and set up pointer on the array:

```c
glGenVertexArrays(1,&vao);
glBindVertexArray(vao);
glGenBuffers(1,&bao);
glBindBuffer(GL_ARRAY_BUFFER,bao);
glBufferData(GL_ARRAY_BUFFER,sizeof(Vertex,&vertexs[0], GL_STATIC_DRAW);
vbo_buffer=glMapBuffer(GL_ARRAY_BUFFER,GL_READ_WRITE);
vertex = glGetAttribLocation(ProgramId,"vertex");
glEnableVertexAttribArray(vertex);
vertexColor = glGetAttribLocation(ProgramId,"vertexColor");
glEnableVertexAttribArray(vertexColor);
vertexNormalID = glGetAttribLocation(ProgramId,"vertexNormal");
glEnableVertexAttribArray(vertexNormalID);
vertexTexID = glGetAttribLocation(ProgramId,"vertexTexel");
glEnableVertexAttribArray(vertexTexID);
glVertexAttribPointer(vertex,4,GL_FLOAT,GL_FALSE,sizeof(vertices[0]),BUFFER_OFFSET(0));
glVertexAttribPointer(vertexColor,4,GL_FLOAT,GL_FALSE,sizeof(vertices[0]),BUFFER_OFFSET(sizeof(vertices[0].vectors)));
glVertexAttribPointer(vertexNormalID,4,GL_FLOAT,GL_FALSE,sizeof(vertices[0]),BUFFER_OFFSET(sizeof(vertices[0].vectors)+sizeof(vertices[0].colors)));
glVertexAttribPointer(vertexTexID,4,GL_FLOAT,GL_FALSE,sizeof(vertices[0].normal));
```

Code 3.7.- Sending mosaic to GPU memory
3.3 Draw mosaic

The mosaic is ready to be drawn. There are two methods of drawing mosaics. The first draws the mosaic from the array which we created, and the second one creates additional arrays with indices pointed to our array. In our project, we will be creating indeces. This method is more intensive, and with it we save more space. To draw many faces, one vertex will be used in a few faces; In this case, we need to repeat this vertex a few times. You should repeat vertices if they have different normals, However, all of our normal are the same therefore, we use indices of vertices, which is much smaller than repeating the vertices. Initialization of the index array is:

```c
vertexIndex = new unsigned int[nindexs*2];
if(vertexIndex==NULL){
    fprintf(stderr,"Error: Init Index Buffer\n");
}
int k=0;
int xoff,yoff,off;
xoff=mborder;
yoff=mborder;
unsigned int ym,xm;
unsigned int ui,uj;
int ij;
```

Code 3.8.- Mosaic indeces
After we initialize our array, we can start creating our indices. The creation of indices includes two parts: creation for tiles and for borders. It is done in this way as it is much easier to apply texture for each tile than to switch textures back and forth for each vertex.

```c
for(ym=0;ym<Yn;ym++){
    yoff=((ym*(mborder+mquad))+mborder)*hstrip;
    for(xm=0;xm<Xn;xm++){
        xoff=(xm*(mborder+mquad))+mborder;
        for(ui=0;ui<mquad-1;ui++){
            for(uj=0;uj<mquad-1;uj++){
                ij=(ui*(hstrip))+uj;
                off=yoff+xoff;
                vertexIndex[k++]=ij+off;
                vertexIndex[k++]=ij+off+1;
                vertexIndex[k++]=ij+hstrip+off;
                vertexIndex[k++]=ij+off+1;
                vertexIndex[k++]=ij+hstrip+off+1;
                vertexIndex[k++]=ij+hstrip+off;
            }
        }
    }
}
```

Code 3.9.- Create mosaic indices

Creating indices are done inside four loops. The first two represent the numbers of the tile row and column base. The last two represent the vertices inside each tile. As you can see, we create six indices for each loop. It is done in this way because OpenGL draws elements using triangles. It takes three vertices and creates a triangle from them. Because we want to draw quads, we use two triangles. Indices for border are done in the same manner
except that we do not need to worry about the different textures that make creating indices much easier. We just check if a vertex is located on a border.

```c
for(i=0;i<stripsx2-1;i++){
    yc=(i)%(mborder+mquad);
    for(j=0;j<hstrip-1;j++){
        ij=(i*(hstrip))+j;
        xc=(j)%(mborder+mquad);
        if( (xc<mborder) | (xc==mborder+mquad-1) | (yc<mborder) |
            (yc==mquad+mborder-1) | (yc<mborder) | (yc==mquad+mborder-1)){
            vertexIndex[k++]=ij;
            vertexIndex[k++]=ij+1;
            vertexIndex[k++]=ij+hstrip;
            vertexIndex[k++]=ij+1;
            vertexIndex[k++]=ij+hstrip;
            vertexIndex[k++]=ij+hstrip+1;
        }
    }
}
```

**Code 3.10.-** Border mosaic indices

After the index is created as in the case of the arrays, we need to create a buffer that will store the indices in GPU memory. However, for arrays of indices, we do not need to create array pointers because our indices are always the same type.

Once the indices are completed, we can now finish drawing our mosaic on a screen.
for(i=0;i<Xn*Yn;i++){
glBindTexture(GL_TEXTURE_2D,pixmaps[mosaicTextureMapping[i+1]].glTexID);
glDrawElements(GL_TRIANGLES,indexPer1Mosaic,GL_UNSIGNED_INT,
BUFFER_OFFSET(sizeof(unsigned int)*(i*indexPer1Mosaic)));

Code 3.11.- Draw mosaic

After all preparations are done, the drawing becomes very simple. We just need to run a command to draw a mosaic by glDrawElements. That function has the following parameters: the type of object we draw using vertices; the count of the indices we draw; type of indices, and the offset to the first vertex we intend to draw. We could draw all the tiles in one row, but we need a loop to switch the texture before drawing the next tile with the glBindTexture function. After that, we draw the borders with one command:

nindexOffset=i*indexPer1Mosaic;
glDrawElements(GL_TRIANGLES,IndexesUsed-nindexOffset,GL_UNSIGNED_INT,
BUFFER_OFFSET(sizeof(unsigned int)*(nindexOffset)));

Code 3.12. – Draw mosaic border

3.4 Light & camera

In this chapter, we will explain how to set up a light and camera in the current environment. To use these objects, we need to set up variables that
will define them and be able to send them to a shader program. For the camera, we need only two structures, a camera location and a camera projection matrix. Earlier, we gave an example of the creation of a projection matrix and its variables to be sent to a shader. Finally to get realistic camera views, we need to multiply the coordinates of the vertex location on the projection matrix:

\[
gl\_Position = \text{ProjectionMatrix} \ast \text{vertex};
\]

Code 3.13. – Apply projection

To define the light source, we need more variables. The first is the location of the light. However, with the light we need the variables that will represent the properties of the light: the shine, the ambient, diffuse, and specular light. After we prepare these variables, we can send them to a shader. We can use them for calculating the color of each vertex. The resulting color will be sum of all the light parameters interacted with the surface. Starting with the ambient light is:

\[
\text{getColor} += \text{vertexColor} \ast \text{LightAmbient};
\]

Code 3.14.- Ambient light
The color of the light comes from an unknown source such as bouncing light from surfaces. We do not use source for it, we just assume that it exists. Next, we need to calculate the diffuse light:

\[
\text{coeficient}=\max(0.0, \text{dot(Normal,lightVector)})
\]
\[
\text{getColor}+= \text{vertexColor} \times \text{LightDiffuse} \times \text{coeficient};
\]

**Code 3.15.- Diffuse light**

Creating a diffuse light is different. Before calculating a resulting color, we need to find an angle between the vertex normal and the light source location. In the specular light, we also have to count the camera location, since the brightness hits the eye:

\[
\text{if} (\text{dot(Normal,lightVector)}>=0.0) \{
\text{getColor}+=\text{LightSpecular} \times \text{vertexColor} \times \text{attenuation} \times \text{pow} (\max (0.0, \text{dot(reflect(-lightVector,Normal),cameraVector)}), 5.0);
\}
\]

**Code 3.16.- Specular light**

Finally we learn all needed steps for creation of the realistic mosaic. Full source code of this project is listed below in appendix b.
Conclusion

The objective of this project was to produce realistic rendering of mosaic grids using GPU programming. We use GLSL shading language to produce results efficiently. GLSL allowed us to interactively create 3D models and manipulate them. OpenGL gives the ability to work with textures that make the creation of realistic objects possible. OpenGL programmable pipeline gives the ability to create new algorithms of vertex and fragment computation that opens tremendous possibilities in the manipulation of 3D objects and the graphic scene around them. It also removes the fixed-function restrictions which existed before. Storing models inside GPU memory gives an improvement in the rendering.

Generating illusions of 3D edges gives an effect that looks very realistic. Testing random bump mapping on tiles results in different types of glass that can also be used in the future for creation of additional mosaic models. The obtained results show that using GLSL is a highly efficient way to create realistic mosaics and interact in real time.
Bibliography


4 OpenGl.org. The Industry's Foundation for High Performance Graphics.
   15 August 2012 <http://www.opengl.org/about/>

5 The freeglut Project. Steve Baker is the official owner/maintainer.
   15 August 2012 <http://freeglut.sourceforge.net/>

6 OpenGL ES - The Standard. The Khronos Group is a not for profit, member-funded consortium focused on the creation of royalty-free open standards for parallel computing, graphics and dynamic media on a wide variety of platforms and devices.
   15 August 2012 <http://www.khronos.org>

7 The Lessons, Learning WebGL. Giles Thomas creator of the web blog.
   15 August 2012 <http://learningwebgl.com/blog/>

8 OpenGL Shading Language @ Lightthource 3D - GLSL Tutorial.
   15 August 2012 <http://openglbook.com>

10  GLSL - Swiftless Tutorials. Donald Urquhart creator of site.

11  Tutorials. Martin Christen creator of website.
    15 August 2012
    <http://www.clockworkcoders.com/oglsl/tutorials.html>
Appendix A. Rendering results

Image 1 - Light effect

Image 2 – Bumped edges without light

Image 3 - Bumps on mosaic

Image 4 – Look in perspective
Image 5 - Flat mosaic

Image 6 – Tile edges

Image 7 - Flat mosaic with texture

Image 8 – Mosaic with edges and texture
Image 9 - High bumps

Image 10 – Intermediate bumps

Image 11- Small bumps

Image 12 – High bumps with small angles for the normals
Image 13- Light in the center

Image 14 – Light in the right corner

Image 15- Light far away right

Image 16 – Light in the left corner
Image 17 - Light on the bottom

Image 18 – Specular reflection

Image 19 - High specular reflection

Image 20 – Diffuse reflection
Image 21 - High diffuse reflection  Image 22 – Specular reflection in perspective

Image 23 - Invisible specular reflection  Image 24 – Perspective without bumps
Image 25- Bottom perspective
Image 26 – Left perspective
Image 27- Right perspective
Image 28 – Top perspective
Appendix B. Program code

Fragment.frag

#version 400
uniform sampler2D colorMap;
uniform int useTexture;
in vec4 pointColor;
in vec4 TexCoord;
out vec4 out_color;
void main(void)
{
    if(useTexture>0){
        if(gl_FrontFacing){
            out_color = pointColor*texture2D(colorMap,TexCoord.st);
        }
        else{
            out_color = pointColor*vec4(0.1,0.1,0.1,1.0);
        }
    }
    else
    
        out_color = pointColor;
}

Vertex.vert

#version 400
uniform mat4 ModelMatrix;
uniform mat4 ViewMatrix;
uniform mat4 ProjectionMatrix;
uniform vec4 Camera;
uniform vec4 LightLocation;
uniform vec4 LightAmbient;
uniform vec4 LightDiffuse;
uniform vec4 LightSpecular;
uniform float LightAttenuation[3];
uniform float SpecularExp;
uniform int AllBlackColor;
in vec4 vertex;
in vec4 vertexColor;
in vec4 vertexNormal;
in vec4 vertexTexel;
out vec4 pointColor;
out vec4 TexCoord;
void main(void)
{
    gl_Position = (ProjectionMatrix*ViewMatrix*ModelMatrix) * vertex;
    if(AllBlackColor==1){
        vec4 Vertex,Normal;
        Vertex = vertex;
        Normal = vertexNormal;
    }
Vertex=ModelMatrix * vertex;
Normal=normalize(ModelMatrix * vertexNormal);
vec4 getColor;
getColor=vec4(0.0,0.0,0.0,1.0);
vec4 lightVector;
lightVector=normalize(LightLocation-Vertex);
vec4 cameraVector;
cameraVector=normalize(Camera-Vertex);
vec4 lightReflectionVector;

lightReflectionVector=normalize(2*dot(lightVector,Normal)*Normal-lightVector);
// attenuation;
float attenuation;
float distance = length(LightLocation-Vertex);
attenuation = 1.0 / (LightAttenuation[0] + LightAttenuation[1]*distance + LightAttenuation[2]*distance*distance);
// attenuation=0.5;
// spotLight;
//--
float coefficient;
// Add ambient light;
getColor += vertexColor*LightAmbient;
// Add diffuse light;
coefficient=max(0.0, dot(Normal,lightVector));
getColor+= vertexColor*LightDiffuse*coefficient;
// Add specular light;
if (dot(Normal,lightVector)>=0.0) {
    getColor+=LightSpecular*vertexColor*attenuation*pow(max(0.0,dot(reflect(-lightVector,Normal),cameraVector)),5.0);
}
coeficient=dot(lightReflectionVector,cameraVector);
if(coefficient>0.0){
    getColor+=
vertexColor*LightSpecular*pow(coeficient,1.0)*SpecularExp;
}
pointColor=getColor;
// pointColor=vertexColor;
}

// pointColor=vertexColor;
// send tex coordinate;
TexCoord = vertexTexel;
Glmosaic.h

#ifndef GLMOSAIC_H
#define GLMOSAIC_H
#include <stdio.h>
#include <math.h>
#include <GL/glew.h>
#include <GL/freeglut.h>
namespace glmosaic{
static const float PI = 3.14159265;
static GLuint ProgramId;
static GLuint VertexShaderId;
static GLuint FragmentShaderId;
typedef struct Vertex{
  GLfloat vectors[4];
  GLfloat colors[4];
  GLfloat normal[4];
  GLfloat texel[4];
} Vertex;

typedef struct Matrix{
  GLfloat m[16];
} Matrix;

typedef struct Light{
  GLfloat location[4];
  GLfloat ambient[4];
  GLfloat diffuse[4];
  GLfloat specular[4];
  GLfloat attenuationCLQ[3];
  GLint locationID;
  GLint ambientID;
  GLint diffuseID;
  GLint specularID;
  GLint attenuationID;
} Light;

GLfloat Cotangent(GLfloat angle);
GLfloat DegreesToRadians(GLfloat degrees);
GLfloat RadiansToDegrees(GLfloat radians);
GLfloat DotProduct3(GLfloat*, GLfloat*);
GLfloat Magnitude3(GLfloat*);
void CrossProduct3(GLfloat*, GLfloat*, GLfloat*);
void NormalizeVector3(GLfloat*);
Matrix MultMatrix(Matrix* m1, Matrix* m2);
void MultMatrixVector(Matrix* m1, Vertex* v2);
Matrix ProjectionMatrix(float fovy, float aspect_ratio, float near_plane, float far_plane);
Matrix OrthoMatrix(GLfloat x1, GLfloat x2, GLfloat y1, GLfloat y2,
  GLfloat z1, GLfloat z2);
void Rotate(Matrix* m1, GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
void RotateVector(Vertex* v1, GLfloat angle, GLfloat x, GLfloat y,
  GLfloat z);
void Scale(Matrix* m1, GLfloat x, GLfloat y, GLfloat z);
void Translate(Matrix* m1, GLfloat x, GLfloat y, GLfloat z);
void PrintMatrix(Matrix* m1);
void Identity(Matrix* m1);
GLuint* CreateShaders(const char* vertexShaderFile, const char* fragmentShaderFile);
void DestroyShaders();
GLint gli(GLint i, GLint j);  //GL use invert indexation.
Function return reversed index in array;
}
#endif // GLMOSAIC_H

Glmosaic.cpp

#include "glmosaic.h"
using namespace glmosaic;
using namespace std;
void CrossProduct3(GLfloat *ret, GLfloat *a, GLfloat *b){
   ret[0]=a[1]*b[2]-a[2]*b[1];
   ret[1]=a[2]*b[0]-a[0]*b[2];
   ret[2]=a[0]*b[1]-a[1]*b[0];
}
GLfloat glmosaic::DotProduct3(GLfloat* a, GLfloat* b){
   GLfloat ret=0;
   int i;
   for(i=0;i<3;i++)
      ret+=a[i]*b[i];
   return ret;
}
GLfloat glmosaic::Magnitude3(GLfloat*a){
   return sqrt((a[0]*a[0])+(a[1]*a[1])+(a[2]*a[2]));
}
void glmosaic::NormalizeVector3(GLfloat *a){
   GLfloat magnitude;
   GLint i;
   magnitude=Magnitude3(a);
   for(i=0;i<3;i++)
      a[i]=a[i]/magnitude;
}
GLfloat glmosaic::RadiansToDegrees(float radians){
   return radians*(GLfloat)(180/PI);
}
GLfloat glmosaic::DegreesToRadians(GLfloat degrees){
   return degrees*(GLfloat)(PI / 180);
}
GLfloat glmosaic::Cotangent(GLfloat angle)
{
   return (GLfloat)(1.0 / tan(angle));
}
GLint glmosaic::gli(GLLint i, GLint j){
   return (4*j+i);
Matrix glmosaic::MultMatrix(Matix *m1, Matix *m2)
{
    Matrix ret;
    int i, j, k;
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < 4; j++)
        {
            ret.m[gli(i, j)] = 0.0;
            for (k = 0; k < 4; k++)
            {
                ret.m[gli(i, j)] += m1->m[gli(i, k)] * m2->m[gli(k, j)];
            }
        }
    }
    return ret;
}

void glmosaic::MultMatrixVector(Matix *m1, Vertex *v2)
{
    Vertex ret;
    int i, j;
    for (i = 0; i < 4; i++)
    {
        ret.normal[i] = 0.0;
        for (j = 0; j < 4; j++)
        {
            ret.normal[i] += m1->m[gli(i, j)] * v2->normal[j];
        }
    }
    for (i = 0; i < 4; i++)
    {
        v2->normal[i] = ret.normal[i];
    }
}

void glmosaic::PrintMatrix(Matix *m1)
{
    int i, j;
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < 4; j++)
        {
            printf("%f2\t", m1->m[gli(i, j)]);
        }
        printf("\n");
    }
}

Matrix glmosaic::OrthoMatrix(GLfloat x2, GLfloat x1, GLfloat y2, GLfloat y1, GLfloat z2, GLfloat z1)
{
    Matrix ret = {{0}};
    ret.m[0] = 2 / (x1 - x2);
    ret.m[5] = 2 / (y1 - y2);
    ret.m[10] = -2 / (z1 - z2);
    //ret.m[12] = -(x1 + x2) / (x1 - x2);
    //ret.m[13] = -(y1 + y2) / (y1 - y2);
    ret.m[14] = -(z1 + z2) / (z1 - z2);
    ret.m[15] = 1.0;
    return ret;
}

Matrix glmosaic::ProjectionMatrix(float fovy, float aspect_ratio, float near_plane, float far_plane)
{
    Matrix ret = { { 0 } };
    const float y_scale = Cotangent(DegreesToRadians(fovy / 2)),
                x_scale = y_scale / aspect_ratio,
                frustum_length = far_plane - near_plane;
    ret.m[0] = x_scale;
ret.m[5] = y_scale;
ret.m[10] = -((far_plane + near_plane) / frustum_length);
ret.m[11] = -1;
ret.m[14] = -(2 * near_plane * far_plane) / frustum_length;
return ret;
}

void glmosaic::Identity(Matrix *m1){
  int i;
  for(i=0;i<16;i++)
    m1->m[i]=0.0;
  m1->m[0]=1.0;
  m1->m[5]=1.0;
  m1->m[10]=1.0;
  m1->m[15]=1.0;
}

void glmosaic::RotateVector(Vertex *v1, GLfloat angle, GLfloat x, GLfloat y, GLfloat z){
  angle = DegreesToRadians(angle);
  Matrix xMatrix = {
    {1.0, 0.0, 0.0, 0.0,
    0.0, cos(angle), -sin(angle), 0.0,
    0.0, sin(angle), cos(angle), 0.0,
    0.0, 0.0, 0.0, 1.0
  }};
  Matrix yMatrix={
    {cos(angle), 0.0, sin(angle), 0.0,
    0.0, 1.0, 0.0, 0.0,
    -sin(angle), 0.0, cos(angle), 0.0,
    0.0, 0.0, 0.0, 1.0}
  };
  Matrix zMatrix={
    {cos(angle),-sin(angle), 0.0, 0.0,
    sin(angle),cos(angle), 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0
  }};
  if(x==1.0){
    MultMatrixVector(&xMatrix,v1);
    NormalizeVector3(v1->normal);
  }
  if(y==1.0){
    MultMatrixVector(&yMatrix,v1);
    NormalizeVector3(v1->normal);
  }
  if(z==1.0){
    MultMatrixVector(&zMatrix,v1);
    NormalizeVector3(v1->normal);
  }
}

void glmosaic::Rotate(Matrix *m1, GLfloat angle, GLfloat x, GLfloat y, GLfloat z){
  angle = DegreesToRadians(angle);
  Matrix xMatrix = {
    {1.0, 0.0, 0.0, 0.0,
    0.0, cos(angle), -sin(angle), 0.0,
    0.0, sin(angle), cos(angle), 0.0,
    0.0, 0.0, 0.0, 1.0
  }};
  Matrix yMatrix={
    {cos(angle), 0.0, sin(angle), 0.0,
    0.0, 1.0, 0.0, 0.0,
    -sin(angle), 0.0, cos(angle), 0.0,
    0.0, 0.0, 0.0, 1.0}
  };
  if(x==1.0){
    MultMatrixVector(&xMatrix,v1);
    NormalizeVector3(v1->normal);
  }
  if(y==1.0){
    MultMatrixVector(&yMatrix,v1);
    NormalizeVector3(v1->normal);
  }
  if(z==1.0){
    MultMatrixVector(&zMatrix,v1);
    NormalizeVector3(v1->normal);
  }
}
Matrix zMatrix={{
    cos(angle), -sin(angle), 0.0, 0.0,
    sin(angle), cos(angle), 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0
}};

Matrix ret;
Identity(&ret);
if (x==1.0)
    ret = MultMatrix(&xMatrix,&ret);
if (y==1.0)
    ret = MultMatrix(&yMatrix,&ret);
if (z==1.0)
    ret = MultMatrix(&zMatrix,&ret);
*m1=MultMatrix(&ret,m1);
}

void glmosaic::Scale(Matrix *m1, GLfloat x, GLfloat y, GLfloat z){
    Matrix ret;
    Identity(&ret);
    ret.m[0]=x;
    ret.m[5]=y;
    ret.m[10]=z;
    *m1=MultMatrix(&ret,m1);
}

void glmosaic::Translate(Matrix *m1, GLfloat x, GLfloat y, GLfloat z){
    Matrix ret;
    Identity(&ret);
    ret.m[12]=x;
    ret.m[13]=y;
    ret.m[14]=z;
    //PrintMatrix(&ret);
    *m1=MultMatrix(&ret,m1);
}

char* ReadShader(const char* shaderFile){
    FILE* fp = fopen(shaderFile,"r");
    if(fp==NULL){return NULL;}
    fseek(fp,0L,SEEK_END);
    long size = ftell(fp);
    fseek(fp,0L,SEEK_SET);
    char* buf = new char[size+1];
    fread(buf,1,size,fp);
    buf[size] = '\0';
    fclose(fp);
    return buf;
}

GLuint* glmosaic::CreateShaders(const char* vertexShaderFile, const char* fragmentShaderFile){
    const GLchar* shaderSource; //shader text;
    GLenum ErrorCheckValue = glGetError(); //Clean error buffer;
    shaderSource = ReadShader(vertexShaderFile); //read vertex shader text from file;
    if(shaderSource==NULL){
        fprintf(stderr,"Failed to read %s\n",vertexShaderFile);
        exit(EXIT_FAILURE);
    }
VertexShaderId = glCreateShader(GL_VERTEX_SHADER); //Create vertex shader id;
glShaderSource(VertexShaderId, 1, &shaderSource, NULL); //upload shader text to video memory;
glCompileShader(VertexShaderId); //Compile vertex shader;
GLint compiled;
glGetShaderiv(VertexShaderId, GL_COMPILE_STATUS, &compiled); //check, did compile complete.
if (!compiled) { //if not write error message and exit;
    printf(stderr, "%s failed to compile: \n", vertexShaderFile);
    GLint logSize;
    glGetShaderiv(VertexShaderId, GL_INFO_LOG_LENGTH, &logSize);
    char* logMsg = new char[logSize];
    glGetShaderInfoLog(VertexShaderId, logSize, NULL, logMsg);
    fprintf(stderr, "%s\n", logMsg);
    delete[] logMsg;
    exit(EXIT_FAILURE);
}
shaderSource = ReadShader(fragmentShaderFile); //read fragment shader text from file;
if (shaderSource == NULL) {
    printf(stderr, "Failed to read %s\n", fragmentShaderFile);
    exit(EXIT_FAILURE);
}
FragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER); //Create fragment shader id;
glShaderSource(FragmentShaderId, 1, &shaderSource, NULL); //upload shader text to video memory;
glCompileShader(FragmentShaderId); //Compile fragment shader;
glGetShaderiv(FragmentShaderId, GL_COMPILE_STATUS, &compiled); //check, did compile complete.
if (!compiled) { //if not write error message and exit;
    printf(stderr, "%s failed to compile: \n", fragmentShaderFile);
    GLint logSize;
    glGetShaderiv(FragmentShaderId, GL_INFO_LOG_LENGTH, &logSize);
    char* logMsg = new char[logSize];
    glGetShaderInfoLog(FragmentShaderId, logSize, NULL, logMsg);
    fprintf(stderr, "%s\n", logMsg);
    delete[] logMsg;
    exit(EXIT_FAILURE);
}
ProgramId = glCreateProgram(); //generate program id;
glAttachShader(ProgramId, VertexShaderId); //attach vertex shader;
glAttachShader(ProgramId, FragmentShaderId); //attach fragment shader;
glLinkProgram(ProgramId); //link program;
GLint linked;
glGetProgramiv(ProgramId, GL_LINK_STATUS, &linked); //get status on link;
if (!linked) { //if link fail give error message and exit;
    printf(stderr, "Shader program failed to link\n\n");
    GLint logSize;
    glGetProgramiv(ProgramId, GL_INFO_LOG_LENGTH, &logSize);
    char* logMsg = new char[logSize];
    glGetProgramInfoLog(ProgramId, logSize, NULL, logMsg);
    fprintf(stderr, "%s\n", logMsg);
}
delete[] logMsg;
exit(EXIT_FAILURE);
}

glUseProgram(ProgramId);  //use program;
ErrorCode = glGetError();  //check is there any errors in
creation of program;
if(ErrorCode != GL_NO_ERROR){
  fprintf(stderr,"Error: Could not create shaders:
%s
",gluErrorString(ErrorCode));
  exit(EXIT_FAILURE);
}

static GLuint ret[3];
ret[0]=ProgramId;
ret[1]=VertexShaderId;
ret[2]=FragmentShaderId;
return ret;

//Function delete shaders;
void glmosaic::DestroyShaders(){
  GLenum ErrorCode = glGetError();  //Clean error buffers;
  glUseProgram(0);  //Turn off shader program;
  glDetachShader(ProgramId,VertexShaderId);
  glDetachShader(ProgramId,FragmentShaderId);
  glDeleteShader(VertexShaderId);
  glDeleteShader(FragmentShaderId);
  glDeleteProgram(ProgramId);
  ErrorCode = glGetError();
  if(ErrorCode!=GL_NO_ERROR){
    fprintf(stderr,"Error: could not destroy the shader:
%s
",gluErrorString(ErrorCode));
    exit(EXIT_FAILURE);
  }
}
```cpp
#include "glmosaic.h"
#include "string.h"
using namespace glmosaic;

//--------Macros;
#define BUFFER_OFFSET( offset ) ((GLvoid*) (offset))
#define WINDOW_TITLE_PREFIX "Mosaic"

// Class read ppm file and store file header: format number, max Colors, width, height and image data.
//
class PPMreader{
public:
    PPMreader(); //init variables;
    GLuint glTexID; //Image id; Each mosaic image has it's number;
    char header[3]; //format number;
    int maxColor; //the maximum colors number;
    int width; //image width;
    int height; //image height;
    GLubyte *data; //image data;

    void ReadFile(char *filename); //read ppm file;
};
PPMreader::PPMreader()
{
    glTexID = 0;
    width = 0;
    height = 0;
    maxColor = 0;
    data = NULL;
}
void PPMreader::ReadFile(char *filename)
{
    FILE *infile;
    char temp;
    infile = fopen(filename,"rb");
    if (!infile){
        fprintf(stderr,"Error: The file, %s, cannot be opened.",filename);
        exit(EXIT_FAILURE);
    }
    //Read file header;
    if (fscanf(infile, "%2s\n", header)==1){ //Read format;
        temp = fgetc(infile);
        if (temp == '#')
            while (temp!='\n')
                temp = fgetc(infile);
        else
            ungetc(temp,infile);
        if (fscanf(infile, "%d %d\n%d\n", &width, &height, &maxColor)==3) //Read width, height, max color;
        {
            data = new GLubyte[width*height*3]; //Init memory for image;
        }
    }
```
if (data == NULL) {
  fprintf(stderr, "Error: ReadFile>data = new...");
  exit(EXIT_FAILURE);
}

fread(data, 3, width * height, infile);  // Read image;
}
else {
  fprintf(stderr, "Error: ReadFile>fscanf(.,&width,...\n"));
  exit(EXIT_FAILURE);
}
else {
  fprintf(stderr, "Problem reading PPM header file - magic number.\n"));
  exit(EXIT_FAILURE);
} fclose(infile);
}  

int WindowID=0;  // glut window id;
int WindowWidth=800;  // glut window width;
int WindowHeight=800;  // glut window height;
void TimerFunction(int);
void MotionFunc(int, int);
void ReshapeFunction(int, int);
void DisplayFunc();
void KeyboardFunc(unsigned char key, int x, int y);
void Cleanup();
void MouseFunc(int, int, int, int);
void PassiveMotionFunc(int, int);
#endif

const char vshader[]="vertex.vert",
  fshader[]="fragment.frag";

//Textures;
PPMreader *pixmaps;  // textures;
int npixmaps=60;  // count of textures;
char mosaicPalette[]="palette.txt";  // texture files name;
char mosaicTexMapping[]="mosaic.txt";  // texture id for tile;
GLint textureShaderID;  // colorMap in fragment shader;
GLint textureOnID;  // use texture in fragment shader;
int textureOn=1;  // texture on, off;

GLint vertex=0,  // vertex shader ID of vertex;
  vertexColor=0,  // vertex shader ID of vertex color; used for light;
  vertexNormalID=0,  // vertex shader ID of vertex normal;
  vertexTexID=0;  // vertex shader ID of texture coordinate;
Matrix modelMatrix,  // world model matrix;
Matrix viewMatrix,  // view matrix;
Matrix projectionMatrix;  // projection Matrix;
GLint modelMatrixID,  // vertex shader ID of model matrix;
  viewMatrixID,  // vertex shader ID of view matrix;
  projectionMatrixID;  // vertex shader ID of projection matrix;
void Init();  // Init shaders, Mosaic, Matrix, light, cameras;
void CreateMosaicVBO(); // Create and move mosaic to video memory;
void DestroyMosaicVBO(); // Free resource for mosaic;
// Color;
unsigned int useLightID=0; // vertex shader ID. If 1 use light else use vertex color;
// Mosaic
int mosaicclock=0;  // blocks growing of memory usage;
GLenum *vbo_buffer;
Matrix mosaicModelMatrix;  // mosaic model matrix;
GLuint vao=0,  // mosaic vertex array object;
bao=0; // mosaic buffer array object;
GLuint mquad=8,  // count of vertex in tile one line;
mborder=2;  // count of vertex in one border between mosaic;
GLuint Xn=71,  // count of mosaic in row;
Yn=71;  // count of mosaic in column;
GLfloat mesh=0.1,  // distance between vertex in tile;
bmesh=0.1;  // distance between vertex in bord;
GLuint indexPer1Mosaic=(mquad-1)*(mquad-1)*6;  // count of indices needed for tile;
GLuint *mosaicTextureMapping;  // array of tile texture mapping;
GLfloat normalAngle=35;  // angle to create 3d borders in mosaic;
GLuint mosaicIndexID;  // vertex shader index ID;
}

float maxAngle=0;
int maxR=5;
float materialColorlvl=1.0;
void CreateMosaicVBO();  // create mosaic in video memory;
void DrawMosaic();  // draw mosaic on screen;
void InitMosaic();
void ResetMosaicMatrix();
// Rotation;
int rotation=0;  // 0-don't rotate, 1-rotate mosaic when mouse move;
// Move;
int moveon=0; // 0 - don't move, 1 - move mosaic when mouse move;
float moveMod=1.0;  // speed with you move mosaic.
void Move(Matrix *, int, int);  // update mosaic matrix when move mouse.
// Light
Matrix lightModelMatrix;  // ligh model matrix;
int lightMove=0;  // Is light move on mouse move; 0 - don't move, 1 - move;
Light light;  // light information; Location, ambient, defuse, specular and attenuation;
\begin{verbatim}
int updateNormals=1;          //Is normals updated; 0 - don't, 1 - only
border of tile, > 2 all normals;
GLfloat lightOffsetX=0;       //if mosaic was moved offsets help to keep
light offset;                 //is normals updated;
lightOffsetY=0;               //only border of tile, > 2 all normals;
float lmultiplier=100;         //switch size of light when mosaic scale
GLfloat specularExp=5.0;       //changes;
int specularExpID;

void LightInit();             //init light;
//Camera
int cameraMove=0;              //is camera move on mouse move; 0 - don't
move, 1 - move;
float omx1=-2.0,               //Ortho camera coordinates.
      omx2=2.0,
      omy1=-2.0,
      omy2=2.0,
      omz1=-100.0,
      omz2=100.0;
GLfloat camera[4]={0.0,0.0,0.0,1.0};       //camera location;
GLint cameraID;                 //vertex shader camera id;
int gmode=0;

void InitViewMatrix();         //init camrea matrix;
void InitProjectionMatrix();   //init projection;
void CameraInit();             //init camera;
void SetHviewMatrix(int lvl);  //scale mosaic to close view;
void SetSviewMatrix();         //scale mosaic to far view;
//support;
int xOld=0,yOld=0;              //previous mouse coordinate;
void Reset();                  //reset mosaic, light and camera
location;
unsigned FrameCount=0;         //show how many screens program draw
per second;
int GenRandNumber(int range);  //Generate random number rand%range;

void Reset(){
    Identity(&modelMatrix);
    InitViewMatrix();
    InitProjectionMatrix();
    ResetMosaicMatrix();
    CameraInit();
    LightInit();
}

int GenRandNumber(int range){
    if(range!=0)
        return (rand()%range);
    else
        return 0;
}
\end{verbatim}
/******Generate Mosaic******/

// function reads in file id's of tiles and saves them in array;

void ReadMosaicMapping(char*){
  unsigned int i;
  FILE *fread;
  fread=fopen(mosaicTexMapping,"r");
  if(!fread){
    fprintf(stderr,"Error: Fail open file %s\n",mosaicTexMapping);
    exit(EXIT_FAILURE);
  }
  float mosaicSize; //length and height of tile;
  float borderSize; //length and height of border;
  fscanf(fread,"%f",&mosaicSize);
  mesh=mosaicSize/(mquad-1); //distance between vertex in tile =
  length of tile / number of vertex;
  fscanf(fread,"%f",&borderSize);
  bmesh=borderSize/(mborder-1); //distance between vertex in border
  = length of border / number of vertex;
  fscanf(fread,"%i %i",&Xn,&Yn);
  //create tile mapping to texture;
  mosaicTextureMapping=new unsigned int[Xn*Yn];
  if(mosaicTextureMapping==NULL){
    fprintf(stderr,"Error: new mosaicTextureMapping.");
  }
  for(i=0;i<Xn*Yn;i++){
    fscanf(fread,"%i",&mosaicTextureMapping[i]);
  }
  fclose(fread);
}

void ResetMosaicMatrix(){
  Identity(&mosaicModelMatrix);
  float offsetx, offsety;
  offsetx=(float)((Xn*mesh*(mquad-1)) )/2;
  offsety=(float)(mesh*(mquad-1)*Yn)/2;
  Translate(&mosaicModelMatrix,-offsetx,-offsety,0.0);
  Rotate(&mosaicModelMatrix,180,0.0,0.0,1.0);
}

void InitMosaic(){
  ReadMosaicMapping(mosaicTexMapping);
  ResetMosaicMatrix();
}

void DrawMosaic(){
  mosaicModelMatrix=MultMatrix(&modelMatrix,&mosaicModelMatrix);
  //update mosaic matrix with world transformation matrix;
glUniformMatrix4fv(modelMatrixID, 1, false, mosaicModelMatrix.m);
// send mosaic model matrix to vertex shader;
glBindVertexArray(vao);  // bind mosaic vao;
glBindBuffer(GL_ARRAY_BUFFER, bao);  // bind mosaic bao;
unsigned int i;
// draw mosaic;
for (i = 0; i < Xn*Yn; i++) {

    glutexture(GL_TEXTURE_2D, pixmaps[mosaicTextureMapping[i + 1]].glTexID) ;
    // set texture for current tile;
    glDrawElements(GL_TRIANGLES, indexPer1Mosaic, GL_UNSIGNED_INT, BUFFER_OFFSET(sizeof(unsigned int)*(i*indexPer1Mosaic)));
    // draw tile;
}

i--;  // number of tiles;
// draw border;
int nindexOffset;  // pointer to border indices;
  nindexOffset = i*indexPer1Mosaic;
  glDrawElements(GL_TRIANGLES, indicesUsed - nindexOffset, GL_UNSIGNED_INT, BUFFER_OFFSET(sizeof(unsigned int)*(nindexOffset)));
// draw border;

// if any error happened;
int ErrorCheckValue = glGetError();
if (ErrorCheckValue != GL_NO_ERROR) {
    fprintf(stderr,
            "ERROR: Could not create a VBO: %s 
",
            gluErrorString(ErrorCheckValue)
    );
    exit(-1);
}

// function generate or regenerate mosaic with different normals;
//
Vertex* UpdateMosaicVertexNormals() {
    // Create different tiles;
    float d;
    int r;
    Vertex *vertices=NULL;  // vertex buffer;
    if (vertices==NULL) {
        vertices = new Vertex[nvertices];
        if (vertices==NULL) {
            fprintf(stderr,"Error: init mosaic buffer;\n");
            exit(EXIT_FAILURE);
        }
    }

    int hstrip = strip>>1;  // half of the strip; count of vertex in one row;
    int stripsx2=strips<<1;  // double of strips; total rows of vertex;
    int i,j;
    int ij;  // i*(one row)+j;
    unsigned int xc=0, yc=0;  // horizontal & vertical borders detection;
float xgrid=0, ygrid=0;    // offset update for vertex;
for(i=0;i<stripsx2;i++){
    for(j=0;j<hstrip;j++){
        ij=(i*hstrip)+j;
        // set default norma;
        vertices[ij].normal[0]=0.0;
        vertices[ij].normal[1]=0.0;
        vertices[ij].normal[2]=1.0;
        vertices[ij].normal[3]=0.0;
        // mborder - number of vertex per border line, mquad -
        // number of vertex per tile line;
        // mborder+mquad used to detect start of new tile or
        border;
        yc=(i)%((mborder+mquad));    // get point position on
        xc=(j)%((mborder+mquad));    // get point position on
        mborder+mquad
        y direction;
        mborder+mquad
        x direction;
        if( (xc<mborder) | (yc<mborder)){
            // color for border;
            vertices[ij].colors[0]=0.0;
            vertices[ij].colors[1]=0.0;
            vertices[ij].colors[2]=0.0;
            vertices[ij].colors[3]=1.0;
        }
        else{
            // color for tile;
            vertices[ij].colors[0]=materialColorlvl;
            vertices[ij].colors[1]=materialColorlvl;
            vertices[ij].colors[2]=materialColorlvl;
            vertices[ij].colors[3]=materialColorlvl;
        }
        // Update normals;
        if(updateNormals>0){
            // if border update normal to look like border;
            if(xc>=mborder){
                if(yc==mborder)
                    RotateVector(&vertices[ij],-
                        normalAngle,1.0,0.0,0.0);
                else if(yc==mborder+mquad-1)
                    RotateVector(&vertices[ij],normalAngle,1.0,0.0,0.0);
                }
            if(yc>=mborder){
                if(xc==mborder){
                    RotateVector(&vertices[ij],normalAngle,0.0,1.0,0.0);
                }
            }
        }
        // Bump;
        d = (float)GenRandNumber(maxAngle);
        r = GenRandNumber(maxR);
        if(d>0){
if(!((xc>=mborder) &(yc==mborder)) & !((yc>=mborder) & (xc>mborder)) )
switch(r) {
case 1:
  RotateVector(&vertices[ij],-d,1.0,0.0,0.0);
  break;
case 2:
  RotateVector(&vertices[ij],d,1.0,0.0,0.0);
  break;
case 3:
  RotateVector(&vertices[ij],-d,0.0,1.0,0.0);
  break;
case 4:
  RotateVector(&vertices[ij],d,0.0,1.0,0.0);
  break;
default://do nothing;
  break;
}
vertices[ij].vectors[0]=xgrid;
vertices[ij].vectors[1]=ygrid;
vertices[ij].vectors[2]=0.0;
vertices[ij].vectors[3]=1.0;
if((xc<mborder) | (xc==mborder+mquad-1)) {
xgrid+=bmesh;
} else {
xgrid+=mesh;
}
vertices[ij].texel[0]=(xc-mborder)*(1.0/mquad);
if(!yc<mborder)
  vertices[ij].texel[0]=(xc-mborder)*(1.0/mquad);
else
  vertices[ij].texel[0]=0;
if(!yc<mborder)
  vertices[ij].texel[1]=(yc-mborder)*(1.0/mquad);
else
  vertices[ij].texel[1]=0.0;
vertices[ij].texel[2]=0.0;
vertices[ij].texel[3]=0.0;
if((yc<mborder) | (yc==mborder+mquad-1)) {
ygrid+=bmesh;
} else {
ygrid+=mesh;
}
xgrid=0.0;
return vertices;
}
//functions update mosaic;
//
void UpdateMosaicVBO(){
    Vertex *vertices; //vertex buffer;
    int sizeofVertex = sizeof(Vertex)*nvertices;
    vertices=UpdateMosaicVertexNormals();
    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, bao);
    glBufferData(GL_ARRAY_BUFFER, sizeofVertex, &vertices[0],
        GL_DYNAMIC_DRAW);
    delete(vertices);
}
//
//functions init mosaic;
//
void CreateMosaicVBO(){
    //Mosaic Vertex Init;
    Vertex *vertices; //vertex buffer;
    int sizeofVertex = sizeof(Vertex)*nvertices;
    int hstrip = strip>>1; //half of the strip;
    int stripsx2 = strips<<1;
    unsigned int xc,yc;
    vertices=UpdateMosaicVertexNormals();
    //VertexArray;
    glGenVertexArrays(1,&vao);
    glBindVertexArray(vao);
    //Create and initialize a buffer object
    glGenBuffers(1,&bao);
    glBindBuffer(GL_ARRAY_BUFFER, bao);
    glBufferData(GL_ARRAY_BUFFER, sizeofVertex, &vertices[0],
        GL_STATIC_DRAW);
    vbo_buffer = glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);
    if(vbo_buffer==NULL){
        fprintf(stderr,"Error: Create>Update vbo.\n");
    } //set up vertex arrays
    vertex = glGetAttribLocation(ProgramId,"vertex");
    glEnableVertexAttribArray(vertex);
    vertexColor = glGetAttribLocation(ProgramId,"vertexColor");
    glEnableVertexAttribArray(vertexColor);
    vertexNormalID = glGetAttribLocation(ProgramId,"vertexNormal");
    if(vertexNormalID!=-1){
        glEnableVertexAttribArray(vertexNormalID);
    }
    else{fprintf(stderr,"Error:Init>Vertex arrays>vertexNormalID = -1\n");}
    vertexTexID = glGetAttribLocation(ProgramId,"vertexTexel");
    if(vertexTexID!=-1){
        glEnableVertexAttribArray(vertexTexID);
    }
    else{fprintf(stderr,"Error:Init>Texel>vertexTexID = -1\n");}
    glVertexAttribPointer(vertex, 4, GL_FLOAT, GL_FALSE, sizeof(vertices[0]), BU
    FFER_OFFSET(0));
    glVertexAttribPointer(vertexColor, 4, GL_FLOAT, GL_FALSE, sizeof(vertices[0]
    ), BUFFER_OFFSET(sizeof(vertices[0].vectors)));
}
glVertexAttribPointer(vertexNormalID, 4, GL_FLOAT, GL_FALSE, sizeof(vertices[0]), BUFFER_OFFSET(sizeof(vertices[0].vectors) + sizeof(vertices[0].colors)));  
glVertexAttribPointer(vertexTexID, 4, GL_FLOAT, GL_FALSE, sizeof(vertices[0]), BUFFER_OFFSET(sizeof(vertices[0].vectors) + sizeof(vertices[0].colors) + sizeof(vertices[0].normal)));  
delete(vertices);  
int ErrorCheckValue = glGetError();  
if (ErrorCheckValue != GL_NO_ERROR)  
{  
    fprintf(stderr,  
        "ERROR: Could not create a VBO: %s \n",  
        gluErrorString(ErrorCheckValue)  
    );  
    exit(-1);  
}  

//Mosaic index init;  
vertexIndex = new unsigned int[nindexes*2];    //array of mosaic indexes;  
if(vertexIndex==NULL){  
    fprintf(stderr,"Error: Init Index Buffer\n");  
}  
int k=0;    //indexes counter;  
int xoff,yoff,off;    //vertex offset, x,y coordinate;  
xoff=mborder;  
yoff=mborder;  
unsigned int ym,xm;  
unsigned int ui,uj;  
int ij;  
//mosaic;  
//generate index for tiles;  
for(ym=0;ym<Yn;ym++){  
    yoff=((ym*(mborder+mquad))+mborder)*hstrip;  
    for(xm=0;xm<Xn;xm++){  
        xoff=(xm*(mborder+mquad))+mborder;  
        for(ui=0;ui<mquad-1;ui++){  
            for(uj=0;uj<mquad-1;uj++){  
                ij=(ui*(hstrip))+uj;  
                off=yoff+xoff;  
                vertexIndex[k++]=ij+off;  
                vertexIndex[k++]=ij+1+off;  
                vertexIndex[k++]=ij+hstrip+off;  
                vertexIndex[k++]=ij+1+off;  
                vertexIndex[k++]=ij+hstrip+1+off;  
                vertexIndex[k++]=ij+hstrip+off;  
            }  
        }  
    }  
}  
int i,j;  
//generate index for borders;  
for(i=0;i<stripsx2-1;i++){  
    yc=(i)%((mborder+mquad));    //get point position on mborder+mquad y direction;
for(j=0;j<hstrip-1;j++){ 
   ij=(i*(hstrip))+j; 
   xc=(j)%(mborder+mquad); //get point position on border+quad x direction; 
   if ( (xc<mborder) | (xc==mborder+mquad-1) | (yc<mborder) | (yc==mquad+mborder-1)){ 
      vertexIndex[k++]=ij; 
      vertexIndex[k++]=ij+1; 
      vertexIndex[k++]=ij+hstrip; 
      vertexIndex[k++]=ij+1; 
      vertexIndex[k++]=ij+hstrip; 
      vertexIndex[k++]=ij+hstrip+1; 
   } 
} 
indicesUsed=k; 
glGenBuffers(1,&mosaicIndexID); 
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mosaicIndexID); 
glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(int)*nindexs,vertexIndex,GL_STATIC_DRAW); 
ErrorCheckValue = glGetError(); 
if (ErrorCheckValue != GL_NO_ERROR){ 
   fprintf(stderr, 
      "ERROR: Could not create a VBO: %s \n", 
      gluErrorString(ErrorCheckValue) 
   ); 
   exit(-1); 
} 
delete (vertexIndex); 
} 
void Move(Matrix *matrix, int xdif, int ydif){ 
   float xmove, ymove; 
   float xm, ym; 
   xm = WindowWidth / (omx2-omx1); 
   ym = WindowHeight / (omy2-omy1); 
   xmove = xdif/xm *moveMod; 
   ymove = ydif/ym *moveMod; 
   Translate(matrix,xmove,-ymove,0.0); 
} 
void LightInit(){ 
   light.location[0]=40.0; light.location[1]=15.0;  
```c
void CameraInit()
{
    camera[0]=0.0;
    camera[1]=0.0;
    camera[2]=1.0;
    camera[3]=1.0;
    InitViewMatrix();
}

void SetHviewMatrix(int lvl)
{
    float hScaleX, hScaleY;
    hScaleX = (mborder*lvl+mborder)*bmesh+(mquad*lvl)*mesh;
    hScaleY = (mborder*lvl+mborder)*bmesh+(mquad*lvl)*mesh;
    hScaleX=(omx2-omx1)/hScaleX;
    hScaleY=(omy2-omy1)/hScaleY;
    Scale(&viewMatrix,hScaleX,hScaleY,1.0);
    glUniformMatrix4fv(viewMatrixID,1,false,viewMatrix.m);
}
```
```
```c
void ResetViewMatrix()
{
    Identity(&viewMatrix);
    Translate(&viewMatrix, 0.0, 0.0, -4.0);
}

void InitViewMatrix()
{
    ResetViewMatrix();
    SetHviewMatrix(6);
}

void InitProjectionMatrix()
{
    Identity(&projectionMatrix);
    projectionMatrix=OrthoMatrix(omx1, omx2, omy1, omy2, omz1, omz2);
    projectionMatrix=ProjectionMatrix(80, WindowWidth/WindowHeight, 1.0f, 100.0f);
    //PrintMatrix(&projectionMatrix);
}

void InitTextures()
{
    //open file with texture names;
    FILE *fread;
    fread=fopen(mosaicPalette,"r"); //Open file with mosaic id + filename;
    if(!fread){
        fprintf(stderr,"Error: The file, %s, cannot be opened.",mosaicPalette);
    }
    int i;
    pixmaps=new PPMreader[npixmaps];
    if(pixmaps==NULL){
        fprintf(stderr,"Error: new Pixmap.");
    }
    char *name;
    name = new char[250];
    char path[250];
    int id;
    //read all textures from file above;
    for(i=0;i<npixmaps;i++){
        fscanf(fread,"%i: %s", &id, name);
        strcpy(path,"/home/dima/QT/College/mosaic/mosaicppm/");
        strcat(path,name);
        pixmaps[i].ReadFile(path);
        glGenTextures(1, &pixmaps[i].g1TexID);
        glBindTexture(GL_TEXTURE_2D, pixmaps[i].g1TexID);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, pixmaps[i].width, pixmaps[i].height, 0, GL_RGB, GL_UNSIGNED_BYTE, pixmaps[i].data);
        glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
    }
}```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
}
delete(name);
fclose(fread);
}

/////////////////////////////////////////Init GL//////////////////////////////////////////

void Init(){
    Identity(&modelMatrix);
    InitViewMatrix();
    InitProjectionMatrix();
    //Load shaders
    GLuint *ids = CreateShaders(vshader, fshader);
    ProgramId = ids[0];
    VertexShaderId = ids[1];
    FragmentShaderId = ids[2];
    //Mosaic;
    InitMosaic();
    CreateMosaicVBO();
    //CreateCube();
    //Matrix;
    modelMatrixID = glGetUniformLocation(ProgramId, "ModelMatrix");
    if(modelMatrixID!=-1){
        glUniformMatrix4fv(modelMatrixID, 1, false, modelMatrix.m);
    }
    viewMatrixID = glGetUniformLocation(ProgramId, "ViewMatrix");
    if(viewMatrixID!=-1){
        glUniformMatrix4fv(viewMatrixID, 1, false, viewMatrix.m);
    }
    projectionMatrixID = glGetUniformLocation(ProgramId, "ProjectionMatrix");
    if(projectionMatrixID!=-1){
        glUniformMatrix4fv(projectionMatrixID, 1, false, projectionMatrix.m);
    }
    //Color;
    useLightID = glGetUniformLocation(ProgramId, "AllBlackColor");
    glUniform1i(useLightID, 1); //use textures;
    //Texture;
    InitTextures();
    textureShaderID=glGetUniformLocation(ProgramId, "colorMap");
    if(textureShaderID!=-1){
        glUniform1i(textureShaderID, 0);
    } else{fprintf(stderr, "Error:Init>Texture>textureShaderID = -1\n");}
    textureOnID=glGetUniformLocation(ProgramId, "useTexture");
    if(textureOnID!=-1){
        glUniform1i(textureOnID, textureOn);
    } else{fprintf(stderr, "Error:Init>Texture>textureOnID = -1\n");}
    //Light;
}
LightInit();
light.locationID = glGetUniformLocation(ProgramId, "LightLocation");
if (light.locationID != -1) {
    glUniform4fv(light.locationID, 1, light.location);
}
light.ambientID = glGetUniformLocation(ProgramId, "LightAmbient");
if (light.ambientID != -1) {
    glUniform4fv(light.ambientID, 1, light.ambient);
} else {
    fprintf(stderr, "Error:Init>Light>ambientID = -1\n");
}
light.diffuseID = glGetUniformLocation(ProgramId, "LightDiffuse");
if (light.diffuseID != -1) {
    glUniform4fv(light.diffuseID, 1, light.diffuse);
} else {
    fprintf(stderr, "Error:Init>Light>diffuseID = -1\n");
}
light.specularID = glGetUniformLocation(ProgramId, "LightSpecular");
if (light.specularID != -1) {
    glUniform4fv(light.specularID, 1, light.specular);
} else {
    fprintf(stderr, "Error:Init>Light>specularID = -1\n");
}
light.attenuationID = glGetUniformLocation(ProgramId, "LightAttenuation");
if (light.attenuationID != -1) {
    glUniform1fv(light.attenuationID, 3, light.attenuationCLQ);
} else {
    fprintf(stderr, "Error:Init>Light>attenuationID = -1\n");
}
specularExpID = glGetUniformLocation(ProgramId, "SpecularExp");
if (specularExpID != -1) {
    glUniform1fv(specularExpID, 1, &specularExp);
} else {
    fprintf(stderr, "Error:Init>Light>specularExpID = -1\n");
}
//Camera;
CameraInit();
cameraID = glGetUniformLocation(ProgramId, "Camera");
if (cameraID != -1) {
    glUniform4fv(cameraID, 1, camera);
} else {
    fprintf(stderr, "Error:Init>Camera>cameraID = -1\n");
}
//SetSviewMatrix();
//GL init;
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glClearColor(0.0, 0.0, 0.5, 1.0);

//-- Destroy Buffers --
void DestroyMosaicVBO()
{
    GLenum ErrorCheckValue = glGetError();
    //delete vbo;
    glDisableVertexAttribArray(vertex);
    glDisableVertexAttribArray(vertexColor);
    glDisableVertexAttribArray(vertexNormalID);
    glDisableVertexAttribArray(vertexTexID);
    glBindBuffer(GL_ARRAY_BUFFER, 0);  //Disconnect from buffers;
    glDeleteBuffers(1, &bao);
    //delete vao;
    glBindVertexArray(0);
    glDeleteVertexArrays(1, &vao);
    ErrorCheckValue = glGetError();
    if (ErrorCheckValue != GL_NO_ERROR) {
        fprintf(stderr, "Error: could not destroy buffers: %s\n", gluErrorString(ErrorCheckValue));
        exit(EXIT_FAILURE);
    }
}
int main(int argc, char **argv) {
    //glut Init
    glutInit(&argc, argv);
    // glutInitContextVersion(4,0);
    // glutInitContextFlags(GLUT_FORWARD_COMPATIBLE);
    // glutInitContextProfile(GLUT_CORE_PROFILE);
    // glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE,GLUT_ACTION_GLUTMAINLOOP_RETURNS);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
    glutInitWindowSize(WindowWidth, WindowHeight);
    WindowID = glutCreateWindow(WINDOW_TITLE_PREFIX);
    if (WindowID < 1) {
        fprintf(stderr, "Error: Could not create a new window.\n");
        exit(EXIT_FAILURE);
    }
    glutDisplayFunc(DisplayFunc);
    glutMotionFunc(MotionFunc);
    glutReshapeFunc(ReshapeFunction);
    glutTimerFunc(0, TimerFunction, 0);
    glutKeyboardFunc(KeyboardFunc);
    glutPassiveMotionFunc(PassiveMotionFunc);
    glutMouseFunc(MouseFunc);
    // glutCloseFunc(Cleanup);
    //glew init;
    GLenum glewInitResult = glewInit();
    if (glewInitResult != GLEW_OK) {
        fprintf(stderr, "Error: %s\n", glewGetErrorString(glewInitResult));
        exit(EXIT_FAILURE);
    }
    //GL init;
    Init();
    glutMainLoop();
    exit(EXIT_SUCCESS);
}

//Function rotate camera if gmode set to 1;
//
int xpOld, ypOld;
void PassiveMotionFunc(int x, int y) {
    float rotate;
    float privX, privY;
    if ((xpOld == 0) || (ypOld == 0)) {
        xpOld = x;
        ypOld = y;
    }
    if (gmode == 1) {
        if (x - xpOld > 0)
            rotate = 1;
        else
            rotate = -1;
        // PrintMatrix(&viewMatrix);
    }
}
//Identity(&viewMatrix);
Rotate(&viewMatrix,-rotate,0,1,0);
// PrintMatrix(&viewMatrix);
glUniformMatrix4fv(viewMatrixID,1,false,viewMatrix.m);
glutPostRedisplay();
if(x>WindowWidth-20){
  glutWarpPointer(21,y);
xpOld=0;
ypOld=0;
}
if(x<20){
  glutWarpPointer(WindowWidth-21,y);
xpOld=0;
ypOld=0;
}
if(y>WindowHeight-20){
  glutWarpPointer(x,21);
xpOld=0;
ypOld=0;
}
if(y<20){
  glutWarpPointer(x,WindowHeight-21);
xpOld=0;
ypOld=0;
}
xpOld=x;
ypOld=y;
}
//Function reset old x,y location;
//
void MouseFunc(int button, int state, int x, int y){
  if(state==GLUT_UP){
    xOld=0;
    yOld=0;
  }
}
//Glut display function;
//
void DisplayFunc(){
  FrameCount++;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  DrawMosaic();
  Identity(&modelMatrix);
  mosaiclock=0;
  glutSwapBuffers();
}
//
//Reshape Window;
//
void ReshapeFunction(int _width, int _height){
  WindowWidth=_width;
  WindowHeight=_height;
  glViewport(0,0,WindowWidth,WindowHeight);
}
// function show performance on window title;
//
void TimerFunction(int Value)
{
    char TempString[512];
    if (0 != Value) {
        sprintf(TempString, "%s: %d Frames Per Second @ %dx%d",
                WINDOW_TITLE_PREFIX, FrameCount * 4, WindowWidth, WindowHeight);
        glutSetWindowTitle(TempString);
    }
    FrameCount = 0;
    glutTimerFunc(250, TimerFunction, 1);
}

// function control from of the program;
//
void KeyboardFunc(unsigned char key, int x, int y){
    GLfloat t=0.1;
    switch(key){
        case 'W':
            case 'w':
                rotation=0;
                moveon=0;
                lightMove=0;
                camera[2]+=1.0;
                Translate(&mosaicModelMatrix, 0.0, 0.0, 1.0);  
                break;
        case 'S':
            case 's':
                rotation=0;
                moveon=0;
                lightMove=0;
                camera[2]+=1.0;
                Translate(&mosaicModelMatrix, 0.0, 0.0, -1.0);  
                break;
        case 'A':
            case 'a':
                rotation=0;
                moveon=0;
                lightMove=0;
                camera[0]+=1.0;
                Translate(&mosaicModelMatrix, 1.0, 0.0, 0.0);  
                break;
        case 'D':
            case 'd':
                rotation=0;
                moveon=0;
                lightMove=0;
                camera[0]+=1.0;
                Translate(&mosaicModelMatrix, -1.0, 0.0, 0.0);  
                break;
        case 'R':
        case 'r':
            rotation++;  
            //Enable rotation
            if(rotation>2)rotation=1;
            lightMove=0;
    }
moveon=0;
gmode=0;
break;
case 'L':
case 'l':   //Enable lighting
    rotation=0;
    moveon=0;
    lightMove=1;
    gmode=0;
    break;
case 'M':
case 'm':
    gmode=0;
    rotation=0;
    lightMove=0;
    moveon=1;
    break;
case 'N':
case 'n':   //Update normals;
    if((mosaiclock==0){
        mosaiclock=1;
        updateNormals=1-updateNormals;
        UpdateMosaicVBO();
    }
    break;
case 'T':
case 't':   //Enable texture;
    textureOn=1-textureOn;
    glUniform1i(textureOnID,textureOn);
    break;
case 'E':
case 'e':
    Reset();   //reset program state;
    break;
case 'H':
case 'h':
    lightMove=0;
    SetLightUp();
    break;
case 'G':
case 'g':
    gmode=1-gmode;
    break;
case '0':
    maxR=5;
    maxAngle=0;
    break;
case '1':
    maxAngle+=2;
    if(maxAngle>18)
        maxAngle=0;
    break;
case '2':
    maxR+=5;
    if(maxR>55){
        maxR=5;
    }
materialColorLvl += 0.1;
if (materialColorLvl > 1.0) {
    materialColorLvl = 0.0;
} break;

case '4':
    static float al = 0.1;
al = al + 0.1;
    if (al > 1.0) {
al = 0;
    }
    light.ambient[3] = 1.0;  //ambient;
    glUniform4fv(light.ambientID, 1, light.ambient);
    break;

case '5':
    static float dl = 0.1;
dl = dl + 0.1;
    if (dl > 1.0) {
dl = 0;
    }
    light.diffuse[0] = dl; light.diffuse[1] = dl;
    glUniform4fv(light.diffuseID, 1, light.diffuse);
    break;

case '7':
    static float sl = 0.1;
    sl = sl + 0.1;
    if (sl > 1.0) {
    sl = 0;
    }
    glUniform4fv(light.specularID, 1, light.specular);
    break;

case '6':
    if (specularExp >= 10) {
        specularExp = 1.0;
    } else {
        specularExp++;
        glUniform1fv(specularExpID, 1, &specularExp);
    } break;

} glutPostRedisplay();

//function does rotation, mosaic move and light move;

void MotionFunc(int x, int y) {
    float speedX, speedY;
    if ((xOld == 0) | (yOld == 0)) {
        xOld = x;
        yOld = y;
    }
    speedX = (x - xOld) >> 1;
speedY = (y - yOld) >> 1;
}
Identity(&modelMatrix);
if((x>0) & (x<WindowWidth) & (y>0) & (y<WindowHeight))return; //if mouse out from window - break;
switch(rotation){
    case 0://rotationSpeed=0;//no rotation;
        Identity(&modelMatrix);
        break;
    case 2: //z rotation;
        if(x!=xOld){
            Rotate(&modelMatrix,speedX,0.0,0.0,1.0);
            glutPostRedisplay();
        }
        break;
    case 1:
        if(y!=yOld){
            Rotate(&modelMatrix,speedY,1.0,0.0,0.0);
        }
        else if(x!=xOld){
            Rotate(&modelMatrix,speedX,0.0,1.0,0.0);
        }
        glutPostRedisplay();
        break;
}
if(lightMove==1){
    float w=(float)WindowWidth/4;
    float h=(float)WindowHeight/4;
    light.location[0]=(float)x/w-2.0+lightOffsetX;
    light.location[1]=(float)-y/h+2.0+lightOffsetY;
    light.location[0]*=lmultiplier;
    light.location[1]*=lmultiplier;
    Identity(&lightModelMatrix);
    Translate(&lightModelMatrix,light.location[0],light.location[1],0.0);
    glUniform4fv(light.locationID,1,light.location);
    glutPostRedisplay();
}
if(moveon==1){
    Move(&mosaicModelMatrix,speedX*10,speedY*10);
    glutPostRedisplay();
}
    xOld=x; yOld=y;
}

//function destroy mosaic shaders and buffers;
// void Cleanup(){
//    DestroyMosaicVBO();
//    DestroyShaders();
//}