FREE eBook

LEARNING

glsl

Free unaffiliated eBook created from
Stack Overflow contributors.

#glsl

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: glsl

It is an unofficial and free glsl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official glsl.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with glsl

Remarks

This section provides an overview of what glsl is, and why a developer might want to use it.

It should also mention any large subjects within glsl, and link out to the related topics. Since the Documentation for glsl is new, you may need to create initial versions of those related topics.

Versions

OpenGL Shading Language

GLSL Version	OpenGL version	Shader Preprocessor	Release Date
1.10	2.0	#version 110	2004-09-07
1.20	2.1	#version 120	2006-07-02
1.30	3.0	#version 130	2008-08-11
1.40	3.1	#version 140	2009-03-24
1.50	3.2	#version 150	2009-08-03
3.30	3.3	#version 330	2010-02-12
4.00	4.0	#version 400	2010-03-11
4.10	4.1	#version 410	2010-07-26
4.20	4.2	#version 420	2011-08-08
4.30	4.3	#version 430	2012-08-06
4.40	4.4	#version 440	2013-07-22
4.50	4.5	#version 450	2014-08-11

OpenGL ES Shading Language

GLSL ES Version	OpenGL ES version	Shader Preprocessor	Release Date
1.00 ES	2.0	#version 100	2007-03-05

GLSL ES Version	OpenGL ES version	Shader Preprocessor	Release Date
3.00 ES	3.0	#version 300 es	2012-08-06
3.10 ES	3.1	#version 310 es	2014-03-17
3.20 ES	3.2	#version 320 es	2015-08-10

Examples

Installation or Setup

Detailed instructions on getting glsl set up or installed.

First OGL 4.0 GLSL shader program

A simple OGL 4.0 GLSL shader program with vertex position and color attribute. The program is executed with a phyton script. To run the script, PyOpenGL must be installed.

A shader program consists at least of a vertex shader and a fragmant shader (exception of computer shaders). The 1st shader stage is the vertex shader and the last shader stage is the fragment shader (In between, optional further stages are possible, which are not further described here).

Vertex shader

first.vet

The vertex shader processes the vertices and associated attributes specified by the drawing command. The vertex shader processes vertices from the input stream and can manipulate it in any desired way. A vertex shader receives one single vertex from the input stream and generates one single vertex to the output vertex stream.

In our example we draw a single triangle, so the vertex shader is executed 3 times, once for each corner point of the triangle. In this case the input to the vertex shader is the vertex position `in vec3 inPos` and the color attribute `in vec3 inCol`. The color attributes is passed to the next shader stage (`out vec3 vertCol`).

```
#version 400

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inCol;

out vec3 vertCol;

void main()
{
    vertCol = inCol;
    gl_Position = vec4( inPos, 1.0 );
```

```
}
```

Fragment shader

first.frag

In this example, the fragment shader follows immediately after the vertex shader. The vertex positions and attributes are interpolated within each face for each fragment. The fragment shader is executed once for each fragment on the entire triangle and receives the color attribute from the frgment shader. Since a triangle is drawn the color attribute is interpolated according to the barycentric coordinates of the fragment based on the drawn triangle.

```
#version 400

in vec3 vertCol;

out vec4 fragColor;

void main()
{
    fragColor = vec4( vertCol, 1.0 );
}
```

Phyton script

The python script is just to compile, link and execute the shader program and to draw geometry. It could be trivially rewritten in C or anything else. It is not the part of this documentation to which the greatest attention should be devoted.

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
from sys import *
from array import array

# draw event
def OnDraw():
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
    glUseProgram( shaderProgram )
    glBindVertexArray( vaObj )
    glDrawArrays( GL_TRIANGLES, 0, 3 )
    glutSwapBuffers()

# read vertex shader program
with open( 'first.vert', 'r' ) as vertFile:
    vertCode = vertFile.read()
print( '\nvertex shader code:' )
print( vertCode )

# read fragment shader program
with open( 'first.frag', 'r' ) as fragFile:
    fragCode = fragFile.read()
print( '\nfragment shader code:' )
```

```
print( fragCode )

# initialize glut
glutInit()

# create window
wndW = 800
wndH = 600
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
glutInitWindowPosition(0, 0)
glutInitWindowSize(wndW, wndH)
wndID = glutCreateWindow(b'OGL window')
glutDisplayFunc(OnDraw)
glutIdleFunc(OnDraw)

# define triangle data
posData = [ -0.636, -0.45, 0.0, 0.636, -0.45, 0.0, 0.0, 0.9, 0.0 ]
colData = [ 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0 ]
posAr = array( "f", posData )
colAr = array( "f", colData )

# create buffers
posBuffer = glGenBuffers(1)
glBindBuffer( GL_ARRAY_BUFFER, posBuffer )
glBufferData( GL_ARRAY_BUFFER, posAr.tostring(), GL_STATIC_DRAW )
colBuffer = glGenBuffers(1)
glBindBuffer( GL_ARRAY_BUFFER, colBuffer )
glBufferData( GL_ARRAY_BUFFER, colAr.tostring(), GL_STATIC_DRAW )

# create vertex array opject
vaObj = glGenVertexArrays( 1 )
glBindVertexArray( vaObj )
glEnableVertexAttribArray( 0 )
glEnableVertexAttribArray( 1 )
glBindBuffer( GL_ARRAY_BUFFER, posBuffer )
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, None )
glBindBuffer( GL_ARRAY_BUFFER, colBuffer )
glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, 0, None )

# compile vertex shader
vertShader = glCreateShader( GL_VERTEX_SHADER )
glShaderSource( vertShader, vertCode )
glCompileShader( vertShader )
result = glGetShaderiv( vertShader, GL_COMPILE_STATUS )
if not (result):
    print( glGetShaderInfoLog( vertShader ) )
    sys.exit()

# compile fragment shader
fragShader = glCreateShader( GL_FRAGMENT_SHADER )
glShaderSource( fragShader, fragCode )
glCompileShader( fragShader )
result = glGetShaderiv( fragShader, GL_COMPILE_STATUS )
if not (result):
    print( glGetShaderInfoLog( fragShader ) )
    sys.exit()

# link shader program
shaderProgram = glCreateProgram()
glAttachShader( shaderProgram, vertShader )
glAttachShader( shaderProgram, fragShader )
```

```
glLinkProgram( shaderProgram )
result = glGetProgramiv( shaderProgram, GL_LINK_STATUS )
if not (result):
    print( 'link error:' )
    print( glGetProgramInfoLog( shaderProgram ) )
    sys.exit()

# start main loop
glutMainLoop()
```

Using of a Model-, View- and Projection matrix in OGL 4.0 GLSL

A simple OGL 4.0 GLSL shader program that shows the use of a model, view, and projection matrix The program is executed with a phyton script. To run the script, PyOpenGL and NumPy must be installed.

- Projection matrix: The projection matrix describes the mapping of a pinhole camera from 3D points in the world to 2D points of the viewport. In this example we use a projection matrix with a field of view of 90 degrees.
- View matrix: The view matrix defines the *eye* position and the viewing direction on the scene. In this example we are moving circular around the scene keeping a viewing direction to the center of the scene.
- Model matrix: The model matrix defines the location and the relative size of an object in the scene. In this example the model matrices move the objects up and down.

Vertex shader

mvp.vet

```
#version 400

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inCol;

out vec3 vertCol;

uniform mat4 projectionMat44;
uniform mat4 viewMat44;
uniform mat4 modelMat44;

void main()
{
    vertCol = inCol;
    vec4 modolPos = modelMat44 * vec4( inPos, 1.0 );
    vec4 viewPos = viewMat44 * modolPos;
    gl_Position = projectionMat44 * viewPos;
}
```

Fragment shader

mvp.frag

```
#version 400

in vec3 vertCol;

out vec4 fragColor;

void main()
{
    fragColor = vec4( vertCol, 1.0 );
}
```

Phyton script

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
from time import time
import math
import sys

# draw event
def OnDraw():
    currentTime = time()
    # set up projection matrix
    prjMat = perspective( 90.0, wndW/wndH, 0.5, 100.0)
    # set up view matrix
    viewMat = Translate( np.matrix(np.identity(4), copy=False, dtype='float32'), np.array(
[0.0, 0.0, -8.0] ) )
    viewMat = RotateView( viewMat, [10.0, CalcAng( currentTime, 10.0 ), 0.0] )

    # set up tetrahedron model matrix
    tetModelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    tetModelMat = RotateX( tetModelMat, -90.0 )
    tetModelMat = Scale( tetModelMat, np.repeat( 2.0, 3 ) )
    tetModelMat = Translate( tetModelMat, np.array( [-2.0, 0.0, CalcMove(currentTime, 6.0, [-
1.0, 1.0])] ) )

    # set up icosahedron model matrix
    icoModelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    icoModelMat = RotateX( icoModelMat, -90.0 )
    icoModelMat = Scale( icoModelMat, np.repeat( 2.0, 3 ) )
    icoModelMat = Translate( icoModelMat, np.array( [2.0, 0.0, CalcMove(currentTime, 6.0,
[1.0, -1.0])] ) )

    # set up attributes and shader program
    glEnable( GL_DEPTH_TEST )
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
    glUseProgram( shaderProgram )
    glUniformMatrix4fv( projectionMatLocation, 1, GL_FALSE, prjMat )
    glUniformMatrix4fv( viewMatLocation, 1, GL_FALSE, viewMat )

    # draw tetrahedron
    glUniformMatrix4fv( modelMatLocation, 1, GL_FALSE, tetModelMat )
    glBindVertexArray( tetVAObj )
    glDrawElements(GL_TRIANGLES, len(tetIndices), GL_UNSIGNED_INT, tetIndices)

    # draw tetrahedron
    glUniformMatrix4fv( modelMatLocation, 1, GL_FALSE, icoModelMat )
```

```
    glBindVertexArray( icoVAObj )
    glDrawArrays( GL_TRIANGLES, 0, len(icoPosData) )

    glutSwapBuffers()

def Fract(val): return val - math.trunc(val)
def CalcAng(currentTime, intervall): return Fract( (currentTime - startTime) / intervall ) *
360.0
def CalcMove(currentTime, intervall, range):
    pos = Fract( (currentTime - startTime) / intervall ) * 2.0
    pos = pos if pos < 1.0 else (2.0-pos)
    return range[0] + (range[1] - range[0]) * pos

# read shader program and compile shader
def CompileShader( sourceFileName, shaderStage ):
    with open( sourceFileName, 'r' ) as sourceFile:
        sourceCode = sourceFile.read()
    nameMap = { GL_VERTEX_SHADER: 'vertex', GL_FRAGMENT_SHADER: 'fragment' }
    print( '\n%s shader code:' % nameMap.get(shaderStage, '') )
    print( sourceCode )
    shaderObj = glCreateShader( shaderStage )
    glShaderSource( shaderObj, sourceCode )
    glCompileShader( shaderObj )
    result = glGetShaderiv( shaderObj, GL_COMPILE_STATUS )
    if not (result):
        print( glGetShaderInfoLog( shaderObj ) )
        sys.exit()
    return shaderObj

# linke shader objects to shader program
def LinkProgram( shaderObjs ):
    shaderProgram = glCreateProgram()
    for shObj in shaderObjs:
        glAttachShader( shaderProgram, shObj )
    glLinkProgram( shaderProgram )
    result = glGetProgramiv( shaderProgram, GL_LINK_STATUS )
    if not (result):
        print( 'link error:' )
        print( glGetProgramInfoLog( shaderProgram ) )
        sys.exit()
    return shaderProgram

# create vertex array opject
def CreateVAO( dataArrays ):
    noOfBuffers = len(dataArrays)
    buffers = glGenBuffers(noOfBuffers)
    newVAObj = glGenVertexArrays( 1 )
    glBindVertexArray( newVAObj )
    for inx in range(0, noOfBuffers):
        vertexSize, dataArr = dataArrays[inx]
        arr = np.array( dataArr, dtype='float32' )
        glBindBuffer( GL_ARRAY_BUFFER, buffers[inx] )
        glBufferData( GL_ARRAY_BUFFER, arr, GL_STATIC_DRAW )
        glEnableVertexAttribArray( inx )
        glVertexAttribPointer( inx, vertexSize, GL_FLOAT, GL_FALSE, 0, None )
    return newVAObj

def Translate(matA, trans):
    matB = np.copy(matA)
    for i in range(0, 4): matB[3,i] = matA[0,i] * trans[0] + matA[1,i] * trans[1] + matA[2,i]
* trans[2] + matA[3,i]
```

```
    return matB

def Scale(matA, s):
    matB = np.copy(matA)
    for i0 in range(0, 3):
        for i1 in range(0, 4): matB[i0,i1] = matA[i0,i1] * s[i0]
    return matB

def RotateHlp(matA, angDeg, a0, a1):
    matB = np.copy(matA)
    ang = math.radians(angDeg)
    sinAng, cosAng = math.sin(ang), math.cos(ang)
    for i in range(0, 4):
        matB[a0,i] = matA[a0,i] * cosAng + matA[a1,i] * sinAng
        matB[a1,i] = matA[a0,i] * -sinAng + matA[a1,i] * cosAng
    return matB

def RotateX(matA, angDeg): return RotateHlp(matA, angDeg, 1, 2)
def RotateY(matA, angDeg): return RotateHlp(matA, angDeg, 2, 0)
def RotateZ(matA, angDeg): return RotateHlp(matA, angDeg, 0, 1)
def RotateView(matA, angDeg): return RotateZ(RotateY(RotateX(matA, angDeg[0]), angDeg[1]),
angDeg[2])

def perspective(fov, aspectRatio, near, far):
    fn, f_n = far + near, far - near
    r, t = aspectRatio, 1.0 / math.tan( math.radians(fov) / 2.0 )
    return np.matrix( [ [t/r,0,0,0], [0,t,0,0], [0,0,-fn/f_n,-2.0*far*near/f_n], [0,0,-1,0] ]
)

# initialize glut
glutInit()

# create window
wndW, wndH = 800, 600
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
glutInitWindowPosition(0, 0)
glutInitWindowSize(wndW, wndH)
wndID = glutCreateWindow(b'OGL window')
glutDisplayFunc(OnDraw)
glutIdleFunc(OnDraw)

# define tetrahedron vertex array opject
sin120 = 0.8660254
tetPposData = [ 0.0, 0.0, 1.0, 0.0, -sin120, -0.5, sin120 * sin120, 0.5 * sin120, -0.5, -
sin120 * sin120, 0.5 * sin120,     -0.5 ]
tetColData = [ 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, ]
tetIndices = [ 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 3, 2 ]
tetVAObj = CreateVAO( [ (3, tetPposData), (3, tetColData) ] )
tetInxArr = np.array( tetIndices, dtype='uint' )

# define icosahedron vertex array opject
icoPts = [
    [ 0.000,  0.000,  1.000], [ 0.894,  0.000,  0.447], [ 0.276,  0.851,  0.447], [-0.724,
0.526,  0.447],
    [-0.724, -0.526,  0.447], [ 0.276, -0.851,  0.447], [ 0.724,  0.526, -0.447], [-0.276,
0.851, -0.447],
    [-0.894,  0.000, -0.447], [-0.276, -0.851, -0.447], [ 0.724, -0.526, -0.447], [ 0.000,
0.000, -1.000] ]
icoCol = [ [1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [1.0, 1.0, 0.0], [0.0, 1.0, 0.0], [1.0, 0.5,
0.0], [1.0, 0.0, 1.0] ]
icoIndices = [
```

```
  2,  0,  1,  3,  0,  2,  4,  0,  3,  5,  0,  4,  1,  0,  5, 11,  7,  6, 11,  8,  7, 11,  9,
8, 11, 10,  9, 11,  6, 10,
  1,  6,  2,  2,  7,  3,  3,  8,  4,  4,  9,  5,  5, 10,  1,  2,  6,  7,  3,  7,  8,  4,  8,
9,  5,  9, 10,  1, 10,  6  ]
icoPosData = []
for inx in icoIndices:
    for inx_s in range(0, 3):
        icoPosData.append( icoPts[inx][inx_s] )
icoColData = []
for inx in range(0, len(icoPosData) // 9):
    inx_col = inx % len(icoCol)
    for inx_p in range(0, 3):
        for inx_s in range(0, 3):
                icoColData.append( icoCol[inx_col][inx_s] )
icoVAObj = CreateVAO( [ (3, icoPosData), (3, icoColData) ] )

# load, compile and link shader
shaderProgram = LinkProgram( [
        CompileShader( 'mvp.vert', GL_VERTEX_SHADER ),
        CompileShader( 'mvp.frag', GL_FRAGMENT_SHADER )
    ] )
projectionMatLocation = glGetUniformLocation(shaderProgram, "projectionMat44")
viewMatLocation = glGetUniformLocation(shaderProgram, "viewMat44")
modelMatLocation = glGetUniformLocation(shaderProgram, "modelMat44")

# start main loop
startTime = time()
glutMainLoop()
```

Put a texture on the model and use a texture matrix in OGL 4.0 GLSL

A simple OGL 4.0 GLSL shader program that shows how to map a 2D texture on a mesh. The program is executed with a phyton script. To run the script, PyOpenGL and NumPy must be installed.

The texture matrix defines how the texture is mapped on the mesh. By manipulating the texture matrix, the texture can be displaced, scaled and rotated.

Vertex shader

tex.vert

#version 400

```
layout (location = 0) in vec3 inPos;
layout (location = 1) in vec2 inTex;

out vec2 vertTex;

uniform mat4 u_projectionMat44;
uniform mat4 u_viewMat44;
uniform mat4 u_modelMat44;
uniform mat4 u_textureMat44;

void main()
{
```

```
    vertTex = ( u_textureMat44 * vec4( inTex, 0.0, 1.0 ) ).st;
    vec4 modolPos = u_modelMat44 * vec4( inPos, 1.0 );
    vec4 viewPos = u_viewMat44 * modolPos;
    gl_Position = u_projectionMat44 * viewPos;
}
```

Fragment shader

tex.frag

```
#version 400

in vec2 vertTex;

out vec4 fragColor;

uniform sampler2D u_texture;

void main()
{
    vec4 texCol = texture( u_texture, vertTex.st );
    fragColor = vec4( texCol.rgb, 1.0 );
}
```

Phyton script

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
from time import time
import math
import sys

# draw event
def OnDraw():
    currentTime = time()
    # set up projection matrix
    prjMat = perspective( 90.0, wndW/wndH, 0.5, 100.0)
    # set up view matrix
    viewMat = Translate( np.matrix(np.identity(4), copy=False, dtype='float32'), np.array(
[0.0, 0.0, -15.0] ) )
    viewMat = RotateView( viewMat, [30.0, CalcAng( currentTime, 60.0 ), 0.0] )

    # set up tetrahedron model matrix
    cubeModelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    cubeModelMat = RotateX( cubeModelMat, -90.0 )
    cubeModelMat = Scale( cubeModelMat, np.repeat( 5.0, 3 ) )

    # set up texture matrix
    texMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    deltaT = Fract( (currentTime - startTime) / 28.0 ) * 28.0
    if deltaT < 7.0 or deltaT >= 21.0:
        texMat = Scale( texMat, np.repeat( CalcMove(currentTime, 7.0, [1.0, 2.0]), 3 ) )
    if deltaT >= 7.0 and deltaT < 14.0 or deltaT >= 21.0:
        transAng = math.radians( CalcAng(currentTime, 7.0) )
        texMat = Translate( texMat, np.array( [math.sin(transAng)*0.5, math.cos(transAng)*0.5-
```

```
0.5, 0.0] ) )
    if deltaT >= 14.0:
        texMat = RotateZ( texMat, CalcAng(currentTime, 7.0) )

    # set up attributes and shader program
    glEnable( GL_DEPTH_TEST )
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
    glUseProgram( shaderProgram )
    glUniformMatrix4fv( projectionMatLocation, 1, GL_FALSE, prjMat )
    glUniformMatrix4fv( viewMatLocation, 1, GL_FALSE, viewMat )
    glUniformMatrix4fv( textureMatLocation, 1, GL_FALSE, texMat )
    glUniform1i( textureLocation, 0 )

    # draw cube
    glUniformMatrix4fv( modelMatLocation, 1, GL_FALSE, cubeModelMat )
    glBindVertexArray( cubeVAObj )
    glDrawElements(GL_TRIANGLES, len(cubeIndices), GL_UNSIGNED_INT, cubeIndices)

    glutSwapBuffers()

def Fract(val): return val - math.trunc(val)
def CalcAng(currentTime, intervall): return Fract( (currentTime - startTime) / intervall ) *
360.0
def CalcMove(currentTime, intervall, range):
    pos = Fract( (currentTime - startTime) / intervall ) * 2.0
    pos = pos if pos < 1.0 else (2.0-pos)
    return range[0] + (range[1] - range[0]) * pos

# read shader program and compile shader
def CompileShader( sourceFileName, shaderStage ):
    with open( sourceFileName, 'r' ) as sourceFile:
        sourceCode = sourceFile.read()
    nameMap = { GL_VERTEX_SHADER: 'vertex', GL_FRAGMENT_SHADER: 'fragment' }
    print( '\n%s shader code:' % nameMap.get(shaderStage, '') )
    print( sourceCode )
    shaderObj = glCreateShader( shaderStage )
    glShaderSource( shaderObj, sourceCode )
    glCompileShader( shaderObj )
    result = glGetShaderiv( shaderObj, GL_COMPILE_STATUS )
    if not (result):
        print( glGetShaderInfoLog( shaderObj ) )
        sys.exit()
    return shaderObj

# linke shader objects to shader program
def LinkProgram( shaderObjs ):
    shaderProgram = glCreateProgram()
    for shObj in shaderObjs:
        glAttachShader( shaderProgram, shObj )
    glLinkProgram( shaderProgram )
    result = glGetProgramiv( shaderProgram, GL_LINK_STATUS )
    if not (result):
        print( 'link error:' )
        print( glGetProgramInfoLog( shaderProgram ) )
        sys.exit()
    return shaderProgram

# create vertex array object
def CreateVAO( dataArrays ):
    noOfBuffers = len(dataArrays)
    buffers = glGenBuffers(noOfBuffers)
```

```
    newVAObj = glGenVertexArrays( 1 )
    glBindVertexArray( newVAObj )
    for inx in range(0, noOfBuffers):
        vertexSize, dataArr = dataArrays[inx]
        arr = np.array( dataArr, dtype='float32' )
        glBindBuffer( GL_ARRAY_BUFFER, buffers[inx] )
        glBufferData( GL_ARRAY_BUFFER, arr, GL_STATIC_DRAW )
        glEnableVertexAttribArray( inx )
        glVertexAttribPointer( inx, vertexSize, GL_FLOAT, GL_FALSE, 0, None )
    return newVAObj

def Translate(matA, trans):
    matB = np.copy(matA)
    for i in range(0, 4): matB[3,i] = matA[0,i] * trans[0] + matA[1,i] * trans[1] + matA[2,i]
* trans[2] + matA[3,i]
    return matB

def Scale(matA, s):
    matB = np.copy(matA)
    for i0 in range(0, 3):
        for i1 in range(0, 4): matB[i0,i1] = matA[i0,i1] * s[i0]
    return matB

def RotateHlp(matA, angDeg, a0, a1):
    matB = np.copy(matA)
    ang = math.radians(angDeg)
    sinAng, cosAng = math.sin(ang), math.cos(ang)
    for i in range(0, 4):
        matB[a0,i] = matA[a0,i] * cosAng + matA[a1,i] * sinAng
        matB[a1,i] = matA[a0,i] * -sinAng + matA[a1,i] * cosAng
    return matB

def RotateX(matA, angDeg): return RotateHlp(matA, angDeg, 1, 2)
def RotateY(matA, angDeg): return RotateHlp(matA, angDeg, 2, 0)
def RotateZ(matA, angDeg): return RotateHlp(matA, angDeg, 0, 1)
def RotateView(matA, angDeg): return RotateZ(RotateY(RotateX(matA, angDeg[0]), angDeg[1]),
angDeg[2])

def perspective(fov, aspectRatio, near, far):
    fn, f_n = far + near, far - near
    r, t = aspectRatio, 1.0 / math.tan( math.radians(fov) / 2.0 )
    return np.matrix( [ [t/r,0,0,0], [0,t,0,0], [0,0,-fn/f_n,-2.0*far*near/f_n], [0,0,-1,0] ]
)

# initialize glut
glutInit()

# create window
wndW, wndH = 800, 600
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
glutInitWindowPosition(0, 0)
glutInitWindowSize(wndW, wndH)
wndID = glutCreateWindow(b'OGL window')
glutDisplayFunc(OnDraw)
glutIdleFunc(OnDraw)

# define cube vertex array opject
icoPts = [
    [-1.0, -1.0,  1.0], [ 1.0, -1.0,  1.0], [ 1.0,  1.0,  1.0], [-1.0,  1.0,  1.0],
    [-1.0, -1.0, -1.0], [ 1.0, -1.0, -1.0], [ 1.0,  1.0, -1.0], [-1.0,  1.0, -1.0] ]
cubePosData = []
```

```
for inx in [ 0, 1, 2, 3, 1, 5, 6, 2, 5, 4, 7, 6, 4, 0, 3, 7, 3, 2, 6, 7, 1, 0, 4, 5 ]:
    for inx_s in range(0, 3): cubePosData.append( icoPts[inx][inx_s] )
cubeTexData = []
for inx in range(0, 6):
    for texCoord in [-0.5, -0.5, 0.5, -0.5, 0.5, 0.5, -0.5, 0.5]: cubeTexData.append( texCoord
)
icoCol = [ [1.0, 0.0, 0.0], [1.0, 0.5, 0.0], [1.0, 0.0, 1.0], [1.0, 1.0, 0.0], [0.0, 1.0,
0.0], [0.0, 0.0, 1.0] ]
cubeIndices = []
for inx in range(0, 6):
    for inx_s in [0, 1, 2, 0, 2, 3]: cubeIndices.append( inx * 4 + inx_s )
cubeVAObj = CreateVAO( [ (3, cubePosData), (2, cubeTexData) ] )
cubeInxArr = np.array( cubeIndices, dtype='uint' )

# load, compile and link shader
shaderProgram = LinkProgram( [
        CompileShader( 'python/ogl4tex/tex.vert', GL_VERTEX_SHADER ),
        CompileShader( 'python/ogl4tex/tex.frag', GL_FRAGMENT_SHADER )
    ] )
projectionMatLocation = glGetUniformLocation(shaderProgram, "u_projectionMat44")
viewMatLocation = glGetUniformLocation(shaderProgram, "u_viewMat44")
modelMatLocation = glGetUniformLocation(shaderProgram, "u_modelMat44")
textureMatLocation = glGetUniformLocation(shaderProgram, "u_textureMat44")
textureLocation = glGetUniformLocation(shaderProgram, "u_texture")

# create texture
texCX, texCY = 128, 128
texPlan = np.zeros( texCX * texCY * 4, dtype=np.uint8 )
for inx_x in range(0, texCX):
    for inx_y in range(0, texCY):
        val_x = math.sin( math.pi * 6.0 * inx_x / texCX )
        val_y = math.sin( math.pi * 6.0 * inx_y / texCY )
        inx_tex = inx_y * texCX * 4 + inx_x * 4
        texPlan[inx_tex + 0] = int( 128 + 127 * val_x )
        texPlan[inx_tex + 1] = 63
        texPlan[inx_tex + 2] = int( 128 + 127 * val_y )
        texPlan[inx_tex + 3] = 255
glActiveTexture( GL_TEXTURE0 )
texObj = glGenTextures( 1  )
glBindTexture( GL_TEXTURE_2D, texObj )
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texCX, texCY, 0, GL_RGBA, GL_UNSIGNED_BYTE, texPlan)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)

# start main loop
startTime = time()
glutMainLoop()
```

Using Interface Block and Uniform Block : A Cook-Torrance light model in OGL 4.0 GLSL

A simple OGL 4.0 GLSL shader program that shows the use of a interface block and a uniform block on a Cook-Torrance microfacet light model implementation. The program is executed with a phyton script. To run the script, PyOpenGL and NumPy must be installed.

An Interface Block is a group of GLSL input, output, uniform, or storage buffer variables. An

Uniform Blockis is an Interface Block with the storage qualifier `uniform`.

Vertex shader

ibub.vert

```
#version 400

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inNV;
layout (location = 2) in vec3 inCol;

out TVertexData
{
    vec3 pos;
    vec3 nv;
    vec3 col;
} outData;

uniform mat4 u_projectionMat44;
uniform mat4 u_modelViewMat44;
uniform mat3 u_normalMat33;

void main()
{
    vec4 viewPos = u_modelViewMat44 * vec4( inPos, 1.0 );

    outData.pos = viewPos.xyz / viewPos.w;
    outData.nv  = u_normalMat33 * normalize( inNV );
    outData.col = inCol;

    gl_Position = u_projectionMat44 * viewPos;
}
```

Fragment shader

ibub.frag

```
#version 400

in TVertexData
{
    vec3 pos;
    vec3 nv;
    vec3 col;
} inData;

out vec4 fragColor;

uniform UB_material
{
    float u_roughness;
    float u_fresnel0;
    vec4  u_specularTint;
};

struct TLightSource
```

```
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 dir;
};

uniform UB_lightSource
{
    TLightSource u_lightSource;
};

vec3 CookTorrance( vec3 esPt, vec3 esPtNV, vec3 col, vec4 specularTint, float roughness, float
fresnel0 )
{
  vec3  esVLight      = normalize( -u_lightSource.dir.xyz );
  vec3  esVEye        = normalize( -esPt );
  vec3  halfVector    = normalize( esVEye + esVLight );
  vec3  reflVector    = normalize( reflect( -esVLight, esPtNV ) );
  float VdotR         = dot( esVEye, reflVector );
  float HdotL         = dot( halfVector, esVLight );
  float NdotL         = dot( esPtNV, esVLight );
  float NdotV         = dot( esPtNV, esVEye );
  float NdotH         = dot( esPtNV, halfVector );
  float NdotH2        = NdotH * NdotH;
  float NdotL_clamped = max( NdotL, 0.0 );
  float NdotV_clamped = max( NdotV, 0.0 );
  float m2            = roughness * roughness;

  // Lambertian diffuse
  float k_diffuse = NdotL_clamped;

  // Cook-Torrance fresnel
  float theta = HdotL;
  float n = (1.0 + sqrt(fresnel0)) / (1.0 - sqrt(fresnel0));
  float g = sqrt( n*n + theta * theta + 1.0 );
  float gc = g + theta;
  float g_c = g - theta;
  float q = (gc * theta - 1.0) / (g_c * theta + 1.0);
  float fresnel = 0.5 * (g_c * g_c) / (gc * gc) * (1.0 + q * q);

  // Gaussian  distribution
  float psi = acos( VdotR );
  float distribution = max( 0.0, HdotL * exp( - psi * psi / m2 ) );

  // Torrance-Sparrow geometric term
  float geometric_att = min( 1.0, min( 2.0 * NdotH * NdotV_clamped / HdotL, 2.0 * NdotH *
NdotL_clamped / HdotL ) );

  // Microfacet bidirectional reflectance distribution function
  float brdf_spec = fresnel * distribution * geometric_att / ( 4.0 * NdotL_clamped *
NdotV_clamped );
  float k_specular = brdf_spec;

  vec3 lightColor = col.rgb * u_lightSource.ambient.rgb
                  + max( 0.0, k_diffuse ) * col.rgb * u_lightSource.diffuse.rgb +
                  + max( 0.0, k_specular ) * mix( col.rgb, specularTint.rgb, specularTint.a )
*     u_lightSource.specular.rgb;
  return lightColor;
}
```

```
void main()
{
    vec3 lightCol = CookTorrance( inData.pos, inData.nv, inData.col, u_specularTint,
u_roughness, u_fresnel0 );
    fragColor = vec4( lightCol, 1.0 );
}
```

Phyton script

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
from time import time
import math
import sys

sin120 = 0.8660254
rotateCamera = False

# draw event
def OnDraw():
    dist = 3.0
    currentTime = time()
    comeraRotAng = CalcAng( currentTime, 10.0 )
    # set up projection matrix
    prjMat = Perspective(90.0, wndW/wndH, 0.5, 100.0)
    # set up view matrix
    viewMat = Translate( np.matrix(np.identity(4), copy=False, dtype='float32'), np.array(
[0.0, 0.0, -12.0] ) )
    viewMat = RotateView( viewMat, [30.0, comeraRotAng if rotateCamera else 0.0, 0.0] )

    # set up light source
    lightSourceBuffer.BindDataFloat(b'u_lightSource.dir', TransformVec4([-3.0, -2.0, -1.0,
0.0], viewMat) )

    # set up tetrahedron model matrix
    tetModelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    if not rotateCamera: tetModelMat = RotateY( tetModelMat, comeraRotAng )
    tetModelMat = RotateX( tetModelMat, -90.0 )
    tetModelMat = Scale( tetModelMat, np.repeat( 2.4, 3 ) )
    tetModelMat = Translate( tetModelMat, np.array( [0.0, dist, 0.0] ) )
    tetModelMat = RotateY( tetModelMat, CalcAng( currentTime, 20.0 ) )
    tetModelMat = RotateX( tetModelMat, CalcAng( currentTime, 9.0 ) )

    # set up icosahedron model matrix
    icoModelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    if not rotateCamera: icoModelMat = RotateY( icoModelMat, comeraRotAng )
    icoModelMat = RotateX( icoModelMat, -90.0 )
    icoModelMat = Scale( icoModelMat, np.repeat( 2.0, 3 ) )
    icoModelMat = Translate( icoModelMat, np.array( [dist * -sin120, dist * -0.5, 0.0] ) )
    icoModelMat = RotateY( icoModelMat, CalcAng( currentTime, 20.0 ) )
    icoModelMat = RotateX( icoModelMat, CalcAng( currentTime, 11.0 ) )

    # set up cube model matrix
    cubeModelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    if not rotateCamera: cubeModelMat = RotateY( cubeModelMat, comeraRotAng )
    cubeModelMat = RotateX( cubeModelMat, -90.0 )
    cubeModelMat = Scale( cubeModelMat, np.repeat( 1.6, 3 ) )
```

```
    cubeModelMat = Translate( cubeModelMat, np.array( [dist * sin120, dist * -0.5, 0.0] ) )
    cubeModelMat = RotateY( cubeModelMat, CalcAng( currentTime, 20.0 ) )
    cubeModelMat = RotateX( cubeModelMat, CalcAng( currentTime, 13.0 ) )

    # set up attributes and shader program
    glEnable( GL_DEPTH_TEST )
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
    glUseProgram( shaderProgram )
    glUniformMatrix4fv( projectionMatLocation, 1, GL_FALSE, prjMat )
    lightSourceBuffer.BindToTarget()

    # draw tetrahedron
    tetMaterialBuffer.BindToTarget()
    modelViewMat = Multiply(viewMat, tetModelMat)
    glUniformMatrix4fv( modelViewMatLocation, 1, GL_FALSE, modelViewMat )
    glUniformMatrix3fv( normalMatLocation, 1, GL_FALSE, ToMat33(modelViewMat) )
    glBindVertexArray( tetVAObj )
    glDrawArrays( GL_TRIANGLES, 0, len(tetPosData) )

    # draw icosahedron
    icoMaterialBuffer.BindToTarget()
    modelViewMat = Multiply(viewMat, icoModelMat)
    glUniformMatrix4fv( modelViewMatLocation, 1, GL_FALSE, modelViewMat )
    glUniformMatrix3fv( normalMatLocation, 1, GL_FALSE, ToMat33(modelViewMat) )
    glBindVertexArray( icoVAObj )
    glDrawArrays( GL_TRIANGLES, 0, len(icoPosData) )

    # draw cube
    cubeMaterialBuffer.BindToTarget()
    modelViewMat = Multiply(viewMat, cubeModelMat)
    glUniformMatrix4fv( modelViewMatLocation, 1, GL_FALSE, modelViewMat )
    glUniformMatrix3fv( normalMatLocation, 1, GL_FALSE, ToMat33(modelViewMat) )
    glBindVertexArray( cubeVAObj )
    glDrawElements(GL_TRIANGLES, len(cubeIndices), GL_UNSIGNED_INT, cubeIndices)

    glutSwapBuffers()

def Fract(val): return val - math.trunc(val)
def CalcAng(currentTime, intervall): return Fract( (currentTime - startTime) / intervall ) *
360.0
def CalcMove(currentTime, intervall, range):
    pos = Fract( (currentTime - startTime) / intervall ) * 2.0
    pos = pos if pos < 1.0 else (2.0-pos)
    return range[0] + (range[1] - range[0]) * pos

# read shader program and compile shader
def CompileShader( sourceFileName, shaderStage ):
    with open( sourceFileName, 'r' ) as sourceFile:
        sourceCode = sourceFile.read()
    nameMap = { GL_VERTEX_SHADER: 'vertex', GL_FRAGMENT_SHADER: 'fragment' }
    print( '\n%s shader code:' % nameMap.get(shaderStage, '') )
    print( sourceCode )
    shaderObj = glCreateShader( shaderStage )
    glShaderSource( shaderObj, sourceCode )
    glCompileShader( shaderObj )
    result = glGetShaderiv( shaderObj, GL_COMPILE_STATUS )
    if not (result):
        print( glGetShaderInfoLog( shaderObj ) )
        sys.exit()
    return shaderObj
```

```
# linke shader objects to shader program
def LinkProgram( shaderObjs ):
    shaderProgram = glCreateProgram()
    for shObj in shaderObjs:
        glAttachShader( shaderProgram, shObj )
    glLinkProgram( shaderProgram )
    result = glGetProgramiv( shaderProgram, GL_LINK_STATUS )
    if not (result):
        print( 'link error:' )
        print( glGetProgramInfoLog( shaderProgram ) )
        sys.exit()
    return shaderProgram

# create vertex array object
def CreateVAO( dataArrays ):
    noOfBuffers = len(dataArrays)
    buffers = glGenBuffers(noOfBuffers)
    newVAObj = glGenVertexArrays( 1 )
    glBindVertexArray( newVAObj )
    for inx in range(0, noOfBuffers):
        vertexSize, dataArr = dataArrays[inx]
        arr = np.array( dataArr, dtype='float32' )
        glBindBuffer( GL_ARRAY_BUFFER, buffers[inx] )
        glBufferData( GL_ARRAY_BUFFER, arr, GL_STATIC_DRAW )
        glEnableVertexAttribArray( inx )
        glVertexAttribPointer( inx, vertexSize, GL_FLOAT, GL_FALSE, 0, None )
    return newVAObj

# representation of a uniform block
class UniformBlock:
     def __init__(self, shaderProg, name):
        self.shaderProg = shaderProg
        self.name = name
     def Link(self, bindingPoint):
        self.bindingPoint = bindingPoint
        self.noOfUniforms = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORMS)
        self.maxUniformNameLen = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORM_MAX_LENGTH)
        self.index = glGetUniformBlockIndex(self.shaderProg, self.name)
        intData = np.zeros(1, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS, intData)
        self.count = intData[0]
        self.indices = np.zeros(self.count, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, self.indices)
        self.offsets = np.zeros(self.count, dtype=int)
        glGetActiveUniformsiv(self.shaderProg, self.count, self.indices, GL_UNIFORM_OFFSET,
self.offsets)
        self.size = 0
        strLengthData = np.zeros(1, dtype=int)
        arraysizeData = np.zeros(1, dtype=int)
        typeData = np.zeros(1, dtype='uint32')
        nameData = np.chararray(self.maxUniformNameLen+1)
        self.namemap = {}
        self.dataSize = 0
        for inx in range(0, len(self.indices)):
            glGetActiveUniform( self.shaderProg, self.indices[inx], self.maxUniformNameLen,
strLengthData, arraysizeData,     typeData, nameData.data )
            name = nameData.tostring()[:strLengthData[0]]
            self.namemap[name] = inx
            self.dataSize = max(self.dataSize, self.offsets[inx] + arraysizeData * 16)
```

```
        glUniformBlockBinding(self.shaderProg, self.index, self.bindingPoint)
        print('\nuniform block %s size:%4d' % (self.name, self.dataSize))
        for uName in self.namemap:
            print( '    %-40s index:%2d    offset:%4d' % (uName,
self.indices[self.namemap[uName]], self.offsets    [self.namemap[uName]]) )

# representation of a uniform block buffer
class UniformBlockBuffer:
    def __init__(self, ub):
        self.namemap = ub.namemap
        self.offsets = ub.offsets
        self.bindingPoint = ub.bindingPoint
        self.object = glGenBuffers(1)
        self.dataSize = ub.dataSize
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.zeros(self.dataSize//4, dtype='float32')
        glBufferData(GL_UNIFORM_BUFFER, self.dataSize, dataArray, GL_DYNAMIC_DRAW)
    def BindToTarget(self):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        glBindBufferBase(GL_UNIFORM_BUFFER, self.bindingPoint, self.object)
    def BindDataFloat(self, name, dataArr):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.array(dataArr, dtype='float32')
        glBufferSubData(GL_UNIFORM_BUFFER, self.offsets[self.namemap[name]], len(dataArr)*4,
dataArray)


def Translate(matA, trans):
    matB = np.copy(matA)
    for i in range(0, 4): matB[3,i] = matA[0,i] * trans[0] + matA[1,i] * trans[1] + matA[2,i]
* trans[2] + matA[3,i]
    return matB

def Scale(matA, s):
    matB = np.copy(matA)
    for i0 in range(0, 3):
        for i1 in range(0, 4): matB[i0,i1] = matA[i0,i1] * s[i0]
    return matB

def RotateHlp(matA, angDeg, a0, a1):
    matB = np.copy(matA)
    ang = math.radians(angDeg)
    sinAng, cosAng = math.sin(ang), math.cos(ang)
    for i in range(0, 4):
        matB[a0,i] = matA[a0,i] * cosAng + matA[a1,i] * sinAng
        matB[a1,i] = matA[a0,i] * -sinAng + matA[a1,i] * cosAng
    return matB

def RotateX(matA, angDeg): return RotateHlp(matA, angDeg, 1, 2)
def RotateY(matA, angDeg): return RotateHlp(matA, angDeg, 2, 0)
def RotateZ(matA, angDeg): return RotateHlp(matA, angDeg, 0, 1)
def RotateView(matA, angDeg): return RotateZ(RotateY(RotateX(matA, angDeg[0]), angDeg[1]),
angDeg[2])

def Multiply(matA, matB):
    matC = np.copy(matA)
    for i0 in range(0, 4):
        for i1 in range(0, 4):
            matC[i0,i1] = matB[i0,0] * matA[0,i1] + matB[i0,1] * matA[1,i1] + matB[i0,2] *
matA[2,i1] + matB[i0,3] * matA    [3,i1]
    return matC
```

```
def ToMat33(mat44):
    mat33 = np.matrix(np.identity(3), copy=False, dtype='float32')
    for i0 in range(0, 3):
        for i1 in range(0, 3): mat33[i0, i1] = mat44[i0, i1]
    return mat33

def TransformVec4(vecA,mat44):
    vecB = np.zeros(4, dtype='float32')
    for i0 in range(0, 4):
        vecB[i0] = vecA[0] * mat44[0,i0] + vecA[1] * mat44[1,i0] + vecA[2] * mat44[2,i0]  +
vecA[3] * mat44[3,i0]
    return vecB

def Perspective(fov, aspectRatio, near, far):
    fn, f_n = far + near, far - near
    r, t = aspectRatio, 1.0 / math.tan( math.radians(fov) / 2.0 )
    return np.matrix( [ [t/r,0,0,0], [0,t,0,0], [0,0,-fn/f_n,-2.0*far*near/f_n], [0,0,-1,0] ]
)

def AddToBuffer( buffer, data, count=1 ):
    for inx_c in range(0, count):
        for inx_s in range(0, len(data)): buffer.append( data[inx_s] )

# initialize glut
glutInit()

# create window
wndW, wndH = 800, 600
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
glutInitWindowPosition(0, 0)
glutInitWindowSize(wndW, wndH)
wndID = glutCreateWindow(b'OGL window')
glutDisplayFunc(OnDraw)
glutIdleFunc(OnDraw)

# define tetrahedron vertex array opject
tetPts = [ (0.0, 0.0, 1.0), (0.0, -sin120, -0.5), (sin120 * sin120, 0.5 * sin120, -0.5), (-
sin120 * sin120, 0.5 * sin120,     -0.5) ]
tetCol = [ [1.0, 0.0, 0.0], [1.0, 1.0, 0.0], [0.0, 0.0, 1.0], [0.0, 1.0, 0.0], ]
tetInxdices = [ 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 3, 2 ]
tetPosData = []
for inx in tetInxdices: AddToBuffer( tetPosData, tetPts[inx] )
tetNVData = []
for inx_nv in range(0, len(tetInxdices) // 3):
    nv = [0.0, 0.0, 0.0]
    for inx_p in range(0, 3):
        for inx_s in range(0, 3): nv[inx_s] += tetPts[ tetInxdices[inx_nv*3 + inx_p] ][inx_s]
    AddToBuffer( tetNVData, nv, 3 )
tetColData = []
for inx_col in range(0, len(tetInxdices) // 3): AddToBuffer( tetColData, tetCol[inx_col %
len(tetCol)], 3 )
tetVAObj = CreateVAO( [ (3, tetPosData), (3, tetNVData), (3, tetColData) ] )

# define icosahedron vertex array opject
icoPts = [
    ( 0.000,  0.000,  1.000), ( 0.894,  0.000,  0.447), ( 0.276,  0.851,  0.447), (-0.724,
0.526,  0.447),
    (-0.724, -0.526,  0.447), ( 0.276, -0.851,  0.447), ( 0.724,  0.526, -0.447), (-0.276,
0.851, -0.447),
    (-0.894,  0.000, -0.447), (-0.276, -0.851, -0.447), ( 0.724, -0.526, -0.447), ( 0.000,
```

```
0.000, -1.000) ]
icoCol = [ [1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [1.0, 1.0, 0.0], [0.0, 1.0, 0.0], [1.0, 0.5,
0.0], [1.0, 0.0, 1.0] ]
icoIndices = [
  2,  0,  1,  3,  0,  2,  4,  0,  3,  5,  0,  4,  1,  0,  5, 11,  7,  6, 11,  8,  7, 11,  9,
8, 11, 10,  9, 11,  6, 10,
  1,  6,  2,  2,  7,  3,  3,  8,  4,  4,  9,  5,  5, 10,  1,  2,  6,  7,  3,  7,  8,  4,  8,
9,  5,  9, 10,  1, 10,  6  ]
icoPosData = []
for inx in icoIndices: AddToBuffer( icoPosData, icoPts[inx] )
icoNVData = []
for inx in icoIndices: AddToBuffer( icoNVData, icoPts[inx] )
#for inx_nv in range(0, len(icoIndices) // 3):
#    nv = [0.0, 0.0, 0.0]
#    for inx_p in range(0, 3):
#        for inx_s in range(0, 3): nv[inx_s] += icoPts[ icoIndices[inx_nv*3 + inx_p] ][inx_s]
#    AddToBuffer( icoNVData, nv, 3 )
icoColData = []
for inx_col in range(0, len(icoIndices) // 3): AddToBuffer( icoColData, icoCol[inx_col %
len(icoCol)], 3 )
icoVAObj = CreateVAO( [ (3, icoPosData), (3, icoNVData), (3, icoColData) ] )

# define cube vertex array opject
cubePts = [
    (-1.0, -1.0,  1.0), ( 1.0, -1.0,  1.0), ( 1.0,  1.0,  1.0), (-1.0,  1.0,  1.0),
    (-1.0, -1.0, -1.0), ( 1.0, -1.0, -1.0), ( 1.0,  1.0, -1.0), (-1.0,  1.0, -1.0) ]
cubeCol = [ [1.0, 0.0, 0.0], [1.0, 0.5, 0.0], [1.0, 0.0, 1.0], [1.0, 1.0, 0.0], [0.0, 1.0,
0.0], [0.0, 0.0, 1.0] ]
cubeHlpInx = [ 0, 1, 2, 3, 1, 5, 6, 2, 5, 4, 7, 6, 4, 0, 3, 7, 3, 2, 6, 7, 1, 0, 4, 5 ]
cubePosData = []
for inx in cubeHlpInx: AddToBuffer( cubePosData, cubePts[inx] )
cubeNVData = []
for inx_nv in range(0, len(cubeHlpInx) // 4):
    nv = [0.0, 0.0, 0.0]
    for inx_p in range(0, 4):
        for inx_s in range(0, 3): nv[inx_s] += cubePts[ cubeHlpInx[inx_nv*4 + inx_p] ][inx_s]
    AddToBuffer( cubeNVData, nv, 4 )
cubeColData = []
for inx_col in range(0, 6):
    AddToBuffer( cubeColData, cubeCol[inx_col % len(cubeCol)], 4 )
cubeIndices = []
for inx in range(0, 6):
    for inx_s in [0, 1, 2, 0, 2, 3]: cubeIndices.append( inx * 4 + inx_s )
cubeVAObj = CreateVAO( [ (3, cubePosData), (3, cubeNVData), (3, cubeColData) ] )

# load, compile and link shader
shaderProgram = LinkProgram( [
        CompileShader( 'ibub.vert', GL_VERTEX_SHADER ),
        CompileShader( 'ibub.frag', GL_FRAGMENT_SHADER )
    ] )
# get unifor locations
projectionMatLocation = glGetUniformLocation(shaderProgram, "u_projectionMat44")
modelViewMatLocation  = glGetUniformLocation(shaderProgram, "u_modelViewMat44")
normalMatLocation     = glGetUniformLocation(shaderProgram, "u_normalMat33")
# linke uniform blocks
ubMaterial = UniformBlock(shaderProgram, "UB_material")
ubLightSource = UniformBlock(shaderProgram, "UB_lightSource")
ubMaterial.Link(1)
ubLightSource.Link(2)

# create uniform block buffers
```

```
lightSourceBuffer = UniformBlockBuffer(ubLightSource)
lightSourceBuffer.BindDataFloat(b'u_lightSource.ambient', [0.1, 0.1, 0.1, 1.0])
lightSourceBuffer.BindDataFloat(b'u_lightSource.diffuse', [0.4, 0.4, 0.4, 1.0])
lightSourceBuffer.BindDataFloat(b'u_lightSource.specular', [1.0, 1.0, 1.0, 1.0])

tetMaterialBuffer = UniformBlockBuffer(ubMaterial)
tetMaterialBuffer.BindDataFloat(b'u_roughness', [0.3])
tetMaterialBuffer.BindDataFloat(b'u_fresnel0', [0.5])
tetMaterialBuffer.BindDataFloat(b'u_specularTint',[1.0, 1.0, 1.0, 0.7])

icoMaterialBuffer = UniformBlockBuffer(ubMaterial)
icoMaterialBuffer.BindDataFloat(b'u_roughness', [0.1])
icoMaterialBuffer.BindDataFloat(b'u_fresnel0', [0.2])
icoMaterialBuffer.BindDataFloat(b'u_specularTint',[1.0, 1.0, 1.0, 0.7])

cubeMaterialBuffer = UniformBlockBuffer(ubMaterial)
cubeMaterialBuffer.BindDataFloat(b'u_roughness', [0.5])
cubeMaterialBuffer.BindDataFloat(b'u_fresnel0', [0.3])
cubeMaterialBuffer.BindDataFloat(b'u_specularTint',[1.0, 1.0, 1.0, 0.7])

# start main loop
startTime = time()
glutMainLoop()
```

Creating geometry using a geometry shader in OGL 4.0 GLSL

A simple OGL 4.0 GLSL shader program that shows the use of geometry shaders. The program is executed with a phyton script. To run the script, PyOpenGL and NumPy must be installed.

In this example, the entire geometry (a cylinder) is generated in the geometry shader.

Vertex shader

geo.vert

```
#version 400

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inNormal;
layout (location = 2) in vec3 inTangent;

out TVertexData
{
    mat3 orientationMat;
} outData;

void main()
{
    vec3 normal   = normalize( inNormal );
    vec3 tangent  = normalize( inTangent );
    vec3 binormal = cross( tangent, normal );

    outData.orientationMat = mat3( normal, cross( binormal, normal ), binormal );
    gl_Position = vec4( inPos, 1.0 );
}
```

Geometry shader

geo.geo

```
#version 400

layout( invocations = 3 ) in;
layout( points ) in;
layout( triangle_strip, max_vertices = 160 ) out;

in TVertexData
{
    mat3 orientationMat;
} inData[];

out TGeometryData
{
    vec3 pos;
    vec3 nv;
    vec3 col;
} outData;

uniform mat4 u_projectionMat44;
uniform mat4 u_viewMat44;
uniform mat4 u_modelMat44;

void NewVertex( in vec3 pt, in mat4 transMat )
{
    vec4 viewPos = transMat * vec4( pt, 1.0 );
    outData.pos = viewPos.xyz / viewPos.w;
    gl_Position = u_projectionMat44 * viewPos;
    EmitVertex();
}

const int circumferenceTile = 36;

void main()
{
    vec4 origin = gl_in[0].gl_Position;
    origin /= origin.w;
    mat4 orintationMat = mat4( vec4( inData[0].orientationMat[0], 0.0 ),
                               vec4( inData[0].orientationMat[1], 0.0 ),
                               vec4( inData[0].orientationMat[2], 0.0 ),
                               origin );
    mat4 modelViewMat = u_viewMat44 * u_modelMat44 * orintationMat;
    mat3 normalMat = mat3( modelViewMat );

    outData.col = vec3( 0.5, 0.7, 0.6 );

    if ( gl_InvocationID == 0 ) // top of the cylinder
    {
        outData.nv  = normalMat * vec3(0.0, 0.0, 1.0);
        vec2 prevPt = vec2( 0.0, 1.0 );
        for ( int inx = 1; inx <= circumferenceTile; inx += 2 )
        {
            float ang1 = 2.0 * 3.14159 * float(inx) / float(circumferenceTile);
            float ang2 = 2.0 * 3.14159 * float(inx+1) / float(circumferenceTile);
            vec2 actPt1 = vec2( sin(ang1), cos(ang1) );
            vec2 actPt2 = vec2( sin(ang2), cos(ang2) );
```

```
            NewVertex( vec3(prevPt.xy, 1.0), modelViewMat );
            NewVertex( vec3(actPt1.xy, 1.0), modelViewMat );
            NewVertex( vec3(0.0, 0.0, 1.0), modelViewMat );
            NewVertex( vec3(actPt2.xy, 1.0), modelViewMat );

            EndPrimitive();
            prevPt = actPt2;
        }
    }

    if ( gl_InvocationID == 1 ) // bottom of the cylinder
    {
        outData.nv  = normalMat * vec3(0.0, 0.0, -1.0);
        vec2 prevPt = vec2( 0.0, 1.0 );
        for ( int inx = circumferenceTile-1; inx >= 0; inx -= 2 )
        {
            float ang1 = 2.0 * 3.14159 * float(inx) / float(circumferenceTile);
            float ang2 = 2.0 * 3.14159 * float(inx-1) / float(circumferenceTile);
            vec2 actPt1 = vec2( sin(ang1), cos(ang1) );
            vec2 actPt2 = vec2( sin(ang2), cos(ang2) );
            NewVertex( vec3(prevPt.xy, -1.0), modelViewMat );
            NewVertex( vec3(actPt1.xy, -1.0), modelViewMat );
            NewVertex( vec3(0.0, 0.0, -1.0), modelViewMat );
            NewVertex( vec3(actPt2.xy, -1.0), modelViewMat );

            EndPrimitive();
            prevPt = actPt2;
        }
    }

    if ( gl_InvocationID == 2 ) // hull of the cylinder
    {
        vec2 prevPt = vec2( 0.0, 1.0 );
        for ( int inx = 1; inx <= circumferenceTile; ++ inx )
        {
            float ang = 2.0 * 3.14159 * float(inx) / float(circumferenceTile);
            vec2 actPt = vec2( sin(ang), cos(ang) );

            outData.nv = normalMat * vec3(prevPt, 0.0);
            NewVertex( vec3(prevPt.xy, -1.0), modelViewMat );
            outData.nv = normalMat * vec3(actPt, 0.0);
            NewVertex( vec3(actPt.xy, -1.0), modelViewMat );
            outData.nv = normalMat * vec3(prevPt, 0.0);
            NewVertex( vec3(prevPt.xy, 1.0), modelViewMat );
            outData.nv = normalMat * vec3(actPt, 0.0);
            NewVertex( vec3(actPt.xy, 1.0), modelViewMat );

            prevPt = actPt;
        }
        EndPrimitive();
    }
}
```

Fragment shader

geo.frag

```
#version 400
```

```
in TGeometryData
{
    vec3 pos;
    vec3 nv;
    vec3 col;
} inData;

out vec4 fragColor;

uniform UB_material
{
    float u_roughness;
    float u_fresnel0;
    vec4  u_specularTint;
};

struct TLightSource
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 dir;
};

uniform UB_lightSource
{
    TLightSource u_lightSource;
};

float Fresnel_Schlick( float theta )
{
    float m = clamp( 1.0 - theta, 0.0, 1.0 );
    float m2 = m * m;
    return m2 * m2 * m; // pow( m, 5.0 )
}

vec3 LightModel( vec3 esPt, vec3 esPtNV, vec3 col, vec4 specularTint, float roughness, float
fresnel0 )
{
  vec3  esVLight      = normalize( -u_lightSource.dir.xyz );
  vec3  esVEye        = normalize( -esPt );
  vec3  halfVector    = normalize( esVEye + esVLight );
  float HdotL         = dot( halfVector, esVLight );
  float NdotL         = dot( esPtNV, esVLight );
  float NdotV         = dot( esPtNV, esVEye );
  float NdotH         = dot( esPtNV, halfVector );
  float NdotH2        = NdotH * NdotH;
  float NdotL_clamped = max( NdotL, 0.0 );
  float NdotV_clamped = max( NdotV, 0.0 );
  float m2            = roughness * roughness;

  // Lambertian diffuse
  float k_diffuse = NdotL_clamped;
  // Schlick approximation
  float fresnel = fresnel0 + ( 1.0 - fresnel0 ) * Fresnel_Schlick( HdotL );
  // Beckmann distribution
  float distribution = max( 0.0, exp( ( NdotH2 - 1.0 ) / ( m2 * NdotH2 ) ) / ( 3.14159265 * m2
* NdotH2 * NdotH2 ) );
  // Torrance-Sparrow geometric term
  float geometric_att = min( 1.0, min( 2.0 * NdotH * NdotV_clamped / HdotL, 2.0 * NdotH *
NdotL_clamped / HdotL ) );
```

```
  // Microfacet bidirectional reflectance distribution function
  float k_specular = fresnel * distribution * geometric_att / ( 4.0 * NdotL_clamped *
NdotV_clamped );

  vec3 lightColor = col.rgb * u_lightSource.ambient.rgb +
                    max( 0.0, k_diffuse ) * col.rgb * u_lightSource.diffuse.rgb +
                    max( 0.0, k_specular ) * mix( col.rgb, specularTint.rgb, specularTint.a )
*     u_lightSource.specular.rgb;
  return lightColor;
}

void main()
{
    vec3 lightCol = LightModel( inData.pos, inData.nv, inData.col, u_specularTint,
u_roughness, u_fresnel0 );
    fragColor = vec4( clamp( lightCol, 0.0, 1.0 ), 1.0 );
}
```

Phyton script

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
from time import time
import math
import sys

sin120 = 0.8660254
rotateCamera = False

# draw event
def OnDraw():
    dist = 3.0
    currentTime = time()
    comeraRotAng = CalcAng( currentTime, 10.0 )
    # set up projection matrix
    prjMat = Perspective(90.0, wndW/wndH, 0.5, 100.0)
    # set up view matrix
    viewMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    viewMat = Translate( viewMat, np.array( [0.0, 0.0, -12.0] ) )
    viewMat = RotateView( viewMat, [30.0, comeraRotAng if rotateCamera else 0.0, 0.0] )

    # set up light source
    lightSourceBuffer.BindDataFloat(b'u_lightSource.dir', TransformVec4([-0.1, 1.0, -5.0,
0.0], viewMat) )

    # set up the model matrix
    modelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    if not rotateCamera: modelMat = RotateY( modelMat, comeraRotAng )
    modelMat = Scale( modelMat, np.repeat( 4, 3 ) )
    #modelMat = Translate( modelMat, np.array( [0.0, 0.0, 1.0] ) )
    #modelMat = RotateY( modelMat, CalcAng( currentTime, 20.0 ) )
    modelMat = RotateX( modelMat, CalcAng( currentTime, 9.0 ) )

    # set up attributes and shader program
    glEnable( GL_DEPTH_TEST )
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
    glUseProgram( shaderProgram )
```

```
    glUniformMatrix4fv( projectionMatLocation, 1, GL_FALSE, prjMat )
    glUniformMatrix4fv( viewMatLocation, 1, GL_FALSE, viewMat )
    lightSourceBuffer.BindToTarget()

    # draw point
    materialBuffer.BindToTarget()
    glUniformMatrix4fv( modelMatLocation, 1, GL_FALSE, modelMat )
    glBindVertexArray( pointVAObj )
    glDrawArrays( GL_POINTS, 0, 1 )

    glutSwapBuffers()

def Fract(val): return val - math.trunc(val)
def CalcAng(currentTime, intervall): return Fract( (currentTime - startTime) / intervall ) *
360.0
def CalcMove(currentTime, intervall, range):
    pos = Fract( (currentTime - startTime) / intervall ) * 2.0
    pos = pos if pos < 1.0 else (2.0-pos)
    return range[0] + (range[1] - range[0]) * pos

# read shader program and compile shader
def CompileShader( sourceFileName, shaderStage ):
    with open( sourceFileName, 'r' ) as sourceFile:
        sourceCode = sourceFile.read()
    nameMap = { GL_VERTEX_SHADER: 'vertex', GL_GEOMETRY_SHADER: 'geometry',
GL_FRAGMENT_SHADER: 'fragment' }
    print( '\n%s shader code:' % nameMap.get(shaderStage, '') )
    print( sourceCode )
    shaderObj = glCreateShader( shaderStage )
    glShaderSource( shaderObj, sourceCode )
    glCompileShader( shaderObj )
    result = glGetShaderiv( shaderObj, GL_COMPILE_STATUS )
    if not (result):
        print( glGetShaderInfoLog( shaderObj ) )
        sys.exit()
    return shaderObj

# linke shader objects to shader program
def LinkProgram( shaderObjs ):
    shaderProgram = glCreateProgram()
    for shObj in shaderObjs:
        glAttachShader( shaderProgram, shObj )
    glLinkProgram( shaderProgram )
    result = glGetProgramiv( shaderProgram, GL_LINK_STATUS )
    if not (result):
        print( 'link error:' )
        print( glGetProgramInfoLog( shaderProgram ) )
        sys.exit()
    return shaderProgram

# create vertex array object
def CreateVAO( dataArrays ):
    noOfBuffers = len(dataArrays)
    buffers = glGenBuffers(noOfBuffers)
    newVAObj = glGenVertexArrays( 1 )
    glBindVertexArray( newVAObj )
    for inx in range(0, noOfBuffers):
        vertexSize, dataArr = dataArrays[inx]
        arr = np.array( dataArr, dtype='float32' )
        glBindBuffer( GL_ARRAY_BUFFER, buffers[inx] )
        glBufferData( GL_ARRAY_BUFFER, arr, GL_STATIC_DRAW )
```

```
        glEnableVertexAttribArray( inx )
        glVertexAttribPointer( inx, vertexSize, GL_FLOAT, GL_FALSE, 0, None )
    return newVAObj

# representation of a uniform block
class UniformBlock:
     def __init__(self, shaderProg, name):
        self.shaderProg = shaderProg
        self.name = name
     def Link(self, bindingPoint):
        self.bindingPoint = bindingPoint
        self.noOfUniforms = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORMS)
        self.maxUniformNameLen = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORM_MAX_LENGTH)
        self.index = glGetUniformBlockIndex(self.shaderProg, self.name)
        intData = np.zeros(1, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS, intData)
        self.count = intData[0]
        self.indices = np.zeros(self.count, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, self.indices)
        self.offsets = np.zeros(self.count, dtype=int)
        glGetActiveUniformsiv(self.shaderProg, self.count, self.indices, GL_UNIFORM_OFFSET,
self.offsets)
        strLengthData = np.zeros(1, dtype=int)
        arraysizeData = np.zeros(1, dtype=int)
        typeData = np.zeros(1, dtype='uint32')
        nameData = np.chararray(self.maxUniformNameLen+1)
        self.namemap = {}
        self.dataSize = 0
        for inx in range(0, len(self.indices)):
            glGetActiveUniform( self.shaderProg, self.indices[inx], self.maxUniformNameLen,
strLengthData, arraysizeData,     typeData, nameData.data )
            name = nameData.tostring()[:strLengthData[0]]
            self.namemap[name] = inx
            self.dataSize = max(self.dataSize, self.offsets[inx] + arraysizeData * 16)
        glUniformBlockBinding(self.shaderProg, self.index, self.bindingPoint)
        print('\nuniform block %s size:%4d' % (self.name, self.dataSize))
        for uName in self.namemap:
            print( '    %-40s index:%2d    offset:%4d' % (uName,
self.indices[self.namemap[uName]], self.offsets    [self.namemap[uName]]) )

# representation of a uniform block buffer
class UniformBlockBuffer:
    def __init__(self, ub):
        self.namemap = ub.namemap
        self.offsets = ub.offsets
        self.bindingPoint = ub.bindingPoint
        self.object = glGenBuffers(1)
        self.dataSize = ub.dataSize
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.zeros(self.dataSize//4, dtype='float32')
        glBufferData(GL_UNIFORM_BUFFER, self.dataSize, dataArray, GL_DYNAMIC_DRAW)
    def BindToTarget(self):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        glBindBufferBase(GL_UNIFORM_BUFFER, self.bindingPoint, self.object)
    def BindDataFloat(self, name, dataArr):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.array(dataArr, dtype='float32')
        glBufferSubData(GL_UNIFORM_BUFFER, self.offsets[self.namemap[name]], len(dataArr)*4,
dataArray)
```

```
def Translate(matA, trans):
    matB = np.copy(matA)
    for i in range(0, 4): matB[3,i] = matA[0,i] * trans[0] + matA[1,i] * trans[1] + matA[2,i]
* trans[2] + matA[3,i]
    return matB

def Scale(matA, s):
    matB = np.copy(matA)
    for i0 in range(0, 3):
        for i1 in range(0, 4): matB[i0,i1] = matA[i0,i1] * s[i0]
    return matB

def RotateHlp(matA, angDeg, a0, a1):
    matB = np.copy(matA)
    ang = math.radians(angDeg)
    sinAng, cosAng = math.sin(ang), math.cos(ang)
    for i in range(0, 4):
        matB[a0,i] = matA[a0,i] * cosAng + matA[a1,i] * sinAng
        matB[a1,i] = matA[a0,i] * -sinAng + matA[a1,i] * cosAng
    return matB

def RotateX(matA, angDeg): return RotateHlp(matA, angDeg, 1, 2)
def RotateY(matA, angDeg): return RotateHlp(matA, angDeg, 2, 0)
def RotateZ(matA, angDeg): return RotateHlp(matA, angDeg, 0, 1)
def RotateView(matA, angDeg): return RotateZ(RotateY(RotateX(matA, angDeg[0]), angDeg[1]),
angDeg[2])

def Multiply(matA, matB):
    matC = np.copy(matA)
    for i0 in range(0, 4):
        for i1 in range(0, 4):
            matC[i0,i1] = matB[i0,0] * matA[0,i1] + matB[i0,1] * matA[1,i1] + matB[i0,2] *
matA[2,i1] + matB[i0,3] * matA    [3,i1]
    return matC

def ToMat33(mat44):
    mat33 = np.matrix(np.identity(3), copy=False, dtype='float32')
    for i0 in range(0, 3):
        for i1 in range(0, 3): mat33[i0, i1] = mat44[i0, i1]
    return mat33

def TransformVec4(vecA,mat44):
    vecB = np.zeros(4, dtype='float32')
    for i0 in range(0, 4):
        vecB[i0] = vecA[0] * mat44[0,i0] + vecA[1] * mat44[1,i0] + vecA[2] * mat44[2,i0]  +
vecA[3] * mat44[3,i0]
    return vecB

def Perspective(fov, aspectRatio, near, far):
    fn, f_n = far + near, far - near
    r, t = aspectRatio, 1.0 / math.tan( math.radians(fov) / 2.0 )
    return np.matrix( [ [t/r,0,0,0], [0,t,0,0], [0,0,-fn/f_n,-2.0*far*near/f_n], [0,0,-1,0] ]
)

def AddToBuffer( buffer, data, count=1 ):
    for inx_c in range(0, count):
        for inx_s in range(0, len(data)): buffer.append( data[inx_s] )

# initialize glut
glutInit()
```

```
# create window
wndW, wndH = 800, 600
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
glutInitWindowPosition(0, 0)
glutInitWindowSize(wndW, wndH)
wndID = glutCreateWindow(b'OGL window')
glutDisplayFunc(OnDraw)
glutIdleFunc(OnDraw)

# define location vertex array opject
pointVAObj = CreateVAO( [ (3, [0.0, 0.0, 0.0] ), (3, [0.0, 0.0, -1.0]), (3, [1.0, 0.0, 0.0]) ]
)

# load, compile and link shader
shaderProgram = LinkProgram( [
        CompileShader( 'geo.vert', GL_VERTEX_SHADER ),
        CompileShader( 'geo.geo', GL_GEOMETRY_SHADER ),
        CompileShader( 'geo.frag', GL_FRAGMENT_SHADER )
    ] )
# get unifor locations
projectionMatLocation = glGetUniformLocation(shaderProgram, "u_projectionMat44")
viewMatLocation       = glGetUniformLocation(shaderProgram, "u_viewMat44")
modelMatLocation      = glGetUniformLocation(shaderProgram, "u_modelMat44")
# linke uniform blocks
ubMaterial = UniformBlock(shaderProgram, "UB_material")
ubLightSource = UniformBlock(shaderProgram, "UB_lightSource")
ubMaterial.Link(1)
ubLightSource.Link(2)

# create uniform block buffers
lightSourceBuffer = UniformBlockBuffer(ubLightSource)
lightSourceBuffer.BindDataFloat(b'u_lightSource.ambient', [0.2, 0.2, 0.2, 1.0])
lightSourceBuffer.BindDataFloat(b'u_lightSource.diffuse', [0.2, 0.2, 0.2, 1.0])
lightSourceBuffer.BindDataFloat(b'u_lightSource.specular', [1.0, 1.0, 1.0, 1.0])

materialBuffer = UniformBlockBuffer(ubMaterial)
materialBuffer.BindDataFloat(b'u_roughness', [0.5])
materialBuffer.BindDataFloat(b'u_fresnel0', [0.2])
materialBuffer.BindDataFloat(b'u_specularTint',[1.0, 0.5, 0.5, 0.8])

# start main loop
startTime = time()
glutMainLoop()
```

Switching the geometry and the surface representation using subroutines in OGL 4.0 GLSL

A simple OGL 4.0 GLSL shader program that shows the use shader subroutines. The program is executed with a phyton script. To run the script, PyOpenGL and NumPy must be installed.

The subroutines switch between different geometry generated in the geometry shader and change the surface representation.

Vertex shader

subr.vert

```
#version 400

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inNormal;
layout (location = 2) in vec3 inTangent;

out TVertexData
{
    mat3 orientationMat;
} outData;

void main()
{
    vec3 normal   = normalize( inNormal );
    vec3 tangent  = normalize( inTangent );
    vec3 binormal = cross( tangent, normal );

    outData.orientationMat = mat3( normal, cross( binormal, normal ), binormal );
    gl_Position = vec4( inPos, 1.0 );
}
```

Geometry shader

subr.geo

```
#version 400

layout( points ) in;
layout( triangle_strip, max_vertices = 512 ) out;

in TVertexData
{
    mat3 orientationMat;
} inData[];

out TGeometryData
{
    vec3 pos;
    vec3 nv;
    vec2 tex;
} outData;

uniform mat4 u_projectionMat44;
uniform mat4 u_viewMat44;
uniform mat4 u_modelMat44;
uniform mat4 u_textureMat44;

void SetTextureCoord( in vec2 tecCoord )
{
    vec4 tex = u_textureMat44 * vec4( tecCoord, 0.0, 1.0 );
    outData.tex = tex.xy;
}

void NewVertex( in vec3 pt, in mat4 transMat )
{
    vec4 viewPos = transMat * vec4( pt, 1.0 );
    outData.pos = viewPos.xyz / viewPos.w;
    gl_Position = u_projectionMat44 * viewPos;
    EmitVertex();
```

```
}

void NewVertexAndTex( in vec3 pt, in mat4 transMat )
{
    SetTextureCoord( pt.xy * 0.5 + 0.5 );
    NewVertex( pt, transMat );
}

void NewVertexNvTex( in vec3 pt, in mat4 transMat, in vec3 nv, in vec2 tex )
{
    outData.nv = nv;
    SetTextureCoord( tex );
    vec4 viewPos = transMat * vec4( pt, 1.0 );
    outData.pos = viewPos.xyz / viewPos.w;
    gl_Position = u_projectionMat44 * viewPos;
    EmitVertex();
}

subroutine void TShape( in mat4 );
subroutine uniform TShape su_shape;

void main()
{
    vec4 origin = gl_in[0].gl_Position;
    origin /= origin.w;
    mat4 orintationMat = mat4( vec4( inData[0].orientationMat[0], 0.0 ),
                               vec4( inData[0].orientationMat[1], 0.0 ),
                               vec4( inData[0].orientationMat[2], 0.0 ),
                               origin );
    mat4 modelMat = u_modelMat44 * orintationMat;

    su_shape( modelMat );
}

subroutine(TShape) void DrawSphere( in mat4 modelMat )
{
    const int circumferenceTile = 18;
    const int layersTile        = 11;

    mat4 modelViewMat = u_viewMat44 * modelMat;
    mat3 normalMat    = mat3( modelViewMat );

    float preStepLay = 0.0;
    vec2  prePtLay   = vec2( 0.0, -1.0 );
    for ( int inxLay = 1; inxLay <= layersTile; ++ inxLay )
    {
        float stepLay = float(inxLay) / float(layersTile);
        float angLay  = 3.14159 * stepLay;
        vec2  ptLay   = vec2( sin(angLay), -cos(angLay) );

        float preStepCir = 0.0;
        vec2  prePtCir   = vec2( 0.0, 1.0 );
        for ( int inxCir = 0; inxCir <= circumferenceTile; ++ inxCir )
        {
            float stepCir = float(inxCir) / float(circumferenceTile);
            float angCir  = 2.0 * 3.14159 * stepCir;
            vec2  ptCir   = vec2( sin(angCir), cos(angCir) );

            if ( inxLay == 1 )
            {
                if ( inxCir >= 0 )
```

```
                {
                    vec3 pt1 = vec3( ptLay.x * prePtCir.x, ptLay.x * prePtCir.y, ptLay.y );
                    vec3 pt2 = vec3( 0.0, 0.0, -1.0 );
                    vec3 pt3 = vec3( ptLay.x * ptCir.x, ptLay.x * ptCir.y, ptLay.y );
                    NewVertexNvTex( pt1, modelViewMat, normalMat * pt1, vec2( preStepCir *
2.0, stepLay ) );
                    NewVertexNvTex( pt2, modelViewMat, normalMat * pt2, vec2( preStepCir +
stepCir, preStepLay )  );
                    NewVertexNvTex( pt3, modelViewMat, normalMat * pt3, vec2( stepCir * 2.0,
stepLay )  );
                    EndPrimitive();
                }
            }
            else if ( inxLay == layersTile )
            {
                if ( inxCir > 0 )
                {
                    vec3 pt1 = vec3( prePtLay.x * prePtCir.x, prePtLay.x * prePtCir.y,
prePtLay.y );
                    vec3 pt2 = vec3( prePtLay.x * ptCir.x, prePtLay.x * ptCir.y, prePtLay.y );
                    vec3 pt3 = vec3( 0.0, 0.0, 1.0 );
                    NewVertexNvTex( pt1, modelViewMat, normalMat * pt1, vec2( preStepCir *
2.0, preStepLay ) );
                    NewVertexNvTex( pt2, modelViewMat, normalMat * pt2, vec2( stepCir * 2.0,
preStepLay )  );
                    NewVertexNvTex( pt3, modelViewMat, normalMat * pt3, vec2( preStepCir +
stepCir, stepLay )  );
                    EndPrimitive();
                }
            }
            else
            {
                vec3 pt1 = vec3( prePtLay.x * ptCir.x, prePtLay.x * ptCir.y, prePtLay.y );
                vec3 pt2 = vec3( ptLay.x * ptCir.x, ptLay.x * ptCir.y, ptLay.y );
                NewVertexNvTex( pt1, modelViewMat, normalMat * pt1, vec2( stepCir * 2.0,
preStepLay ) );
                NewVertexNvTex( pt2, modelViewMat, normalMat * pt2, vec2( stepCir * 2.0,
stepLay )  );
            }

            preStepCir = stepCir;
            prePtCir   = ptCir;
        }
        if ( inxLay > 1 && inxLay < layersTile )
            EndPrimitive();

        preStepLay = stepLay;
        prePtLay   = ptLay;
    }
}

subroutine(TShape) void DrawTorus( in mat4 modelMat )
{
    const int   circumferenceTile = 12;
    const int   layersTile        = 18;
    const float torusRad          = 0.8;
    const float ringRad           = 0.4;

    mat4 modelViewMat = u_viewMat44 * modelMat;
    mat3 normalMat    = mat3( modelViewMat );
```

```
    float preStepLay = 0.0;
    mat4  prePosMat;
    for ( int inxLay = 0; inxLay <= layersTile; ++ inxLay )
    {
        float stepLay = float(inxLay) / float(layersTile);
        float angLay  = 2.0 * 3.14159 * stepLay;
        mat4  posMat = mat4(
            vec4( cos(angLay), sin(angLay), 0.0, 0.0 ),
            vec4( sin(angLay), cos(angLay), 0.0, 0.0 ),
            vec4( 0.0, 0.0, 1.0, 0.0 ),
            vec4( cos(angLay) * torusRad, sin(angLay) * torusRad, 0.0, 1.0 ) );

        for ( int inxCir = 0; inxLay > 0 && inxCir <= circumferenceTile; ++ inxCir )
        {
            float stepCir = float(inxCir) / float(circumferenceTile);
            float angCir  = 2.0 * 3.14159 * stepCir;
            vec2  ptCir   = vec2( sin(angCir), cos(angCir) );

            vec4 tempPt = vec4( ptCir.x * ringRad, 0.0, ptCir.y * ringRad, 1.0 );
            vec4 pt1 = prePosMat * tempPt;
            vec4 pt2 = posMat * tempPt;
            NewVertexNvTex( pt1.xyz, modelViewMat, normalMat * normalize(pt1.xyz -
prePosMat[3].xyz), vec2(stepCir,     preStepLay*2.0) );
            NewVertexNvTex( pt2.xyz, modelViewMat, normalMat * normalize(pt2.xyz -
posMat[3].xyz), vec2(stepCir, stepLay*2.0)      );
        }
        EndPrimitive();

        preStepLay = stepLay;
        prePosMat  = posMat;
    }
}
```

Fragment shader

subr.frag

```
#version 400

in TGeometryData
{
    vec3 pos;
    vec3 nv;
    vec2 tex;
} inData;

out vec4 fragColor;

uniform sampler2D u_texture;

uniform UB_material
{
    float u_roughness;
    float u_fresnel0;
    vec4  u_color;
    vec4  u_specularTint;
};

struct TLightSource
```

```
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 dir;
};

uniform UB_lightSource
{
    TLightSource u_lightSource;
};

subroutine vec4 TSurface( void );
subroutine uniform TSurface su_surface;

float Fresnel_Schlick( in float theta );
vec3 LightModel( in vec3 esPt, in vec3 esPtNV, in vec3 col, in vec4 specularTint, in float
roughness, in float fresnel0 );

void main()
{
    vec4 fragCol = su_surface();
    vec3 lightCol = LightModel( inData.pos, inData.nv, fragCol.rgb, u_specularTint,
u_roughness, u_fresnel0 );

    fragColor = vec4( clamp( lightCol, 0.0, 1.0 ), fragCol.a );
}

subroutine(TSurface) vec4 SurfaceColor( void )
{
  return u_color;
}

subroutine(TSurface) vec4 SurfaceTexture( void )
{
  return texture( u_texture, inData.tex.st );
}

float Fresnel_Schlick( in float theta )
{
    float m = clamp( 1.0 - theta, 0.0, 1.0 );
    float m2 = m * m;
    return m2 * m2 * m; // pow( m, 5.0 )
}

vec3 LightModel( in vec3 esPt, in vec3 esPtNV, in vec3 col, in vec4 specularTint, in float
roughness, in float fresnel0 )
{
  vec3  esVLight      = normalize( -u_lightSource.dir.xyz );
  vec3  esVEye        = normalize( -esPt );
  vec3  halfVector    = normalize( esVEye + esVLight );
  float HdotL         = dot( halfVector, esVLight );
  float NdotL         = dot( esPtNV, esVLight );
  float NdotV         = dot( esPtNV, esVEye );
  float NdotH         = dot( esPtNV, halfVector );
  float NdotH2        = NdotH * NdotH;
  float NdotL_clamped = max( NdotL, 0.0 );
  float NdotV_clamped = max( NdotV, 0.0 );
  float m2            = roughness * roughness;

  // Lambertian diffuse
```

```
  float k_diffuse = NdotL_clamped;
  // Schlick approximation
  float fresnel = fresnel0 + ( 1.0 - fresnel0 ) * Fresnel_Schlick( HdotL );
  // Beckmann distribution
  float distribution = max( 0.0, exp( ( NdotH2 - 1.0 ) / ( m2 * NdotH2 ) ) / ( 3.14159265 * m2
* NdotH2 * NdotH2 ) );
  // Torrance-Sparrow geometric term
  float geometric_att = min( 1.0, min( 2.0 * NdotH * NdotV_clamped / HdotL, 2.0 * NdotH *
NdotL_clamped / HdotL ) );
  // Microfacet bidirectional reflectance distribution function
  float k_specular = fresnel * distribution * geometric_att / ( 4.0 * NdotL_clamped *
NdotV_clamped );

  vec3 lightColor = col.rgb * u_lightSource.ambient.rgb +
                    max( 0.0, k_diffuse ) * col.rgb * u_lightSource.diffuse.rgb +
                    max( 0.0, k_specular ) * mix( col.rgb, specularTint.rgb, specularTint.a )
* u_lightSource.specular.rgb;
  return lightColor;
}
```

Phyton script

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
from time import time
import math
import sys

sin120 = 0.8660254
rotateCamera = False

# draw event
def OnDraw():
    dist = 3.0
    currentTime = time()
    comeraRotAng = CalcAng( currentTime, 10.0 )
    # set up projection matrix
    prjMat = Perspective(90.0, wndW/wndH, 0.5, 100.0)
    # set up view matrix
    viewMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    viewMat = Translate( viewMat, np.array( [0.0, 0.0, -14.0] ) )
    viewMat = RotateView( viewMat, [30.0, comeraRotAng if rotateCamera else 0.0, 0.0] )

    # set up light source
    lightSourceBuffer.BindDataFloat(b'u_lightSource.dir', TransformVec4([-1.0, -1.0, -5.0,
0.0], viewMat) )

    # set up model matrices
    modelMat = []
    for inx in range(0, 2):
        modelMat.append( np.matrix(np.identity(4), copy=False, dtype='float32') )
        if not rotateCamera: modelMat[inx] = RotateY( modelMat[inx], comeraRotAng )

    modelMat[0] = Scale( modelMat[0], np.repeat( 3, 3 ) )
    modelMat[0] = Translate( modelMat[0], np.array( [0.0, 0.0, -2.0] ) )
    modelMat[0] = RotateY( modelMat[0], CalcAng( currentTime, 23.0 ) )
    modelMat[0] = RotateX( modelMat[0], CalcAng( currentTime, 13.0 ) )
```

```
    modelMat[1] = Scale( modelMat[1], np.repeat( 3, 3 ) )
    modelMat[1] = Translate( modelMat[1], np.array( [0.0, 0.0, 2.0] ) )
    modelMat[1] = RotateY( modelMat[1], CalcAng( currentTime, 17.0 ) )
    modelMat[1] = RotateX( modelMat[1], CalcAng( currentTime, 9.0 ) )

    # set up texture matrix
    texMat = np.matrix(np.identity(4), copy=False, dtype='float32')

    # set up attributes and shader program
    glEnable( GL_DEPTH_TEST )
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
    glUseProgram( shaderProgram )
    glUniformMatrix4fv( projectionMatLocation, 1, GL_FALSE, prjMat )
    glUniformMatrix4fv( viewMatLocation, 1, GL_FALSE, viewMat )
    glUniformMatrix4fv( textureMatLocation, 1, GL_FALSE, texMat )
    glUniform1i( textureLocation, 0 )
    lightSourceBuffer.BindToTarget()

    # draw points
    glBindVertexArray( pointVAObj )
    for inx in range(0, 2):
        # set up geometry shader subroutine
        shape = 1 if inx==0 else 0 # 0: sphere, 1: torus
        glUniformSubroutinesuiv(GL_GEOMETRY_SHADER, 1, np.array( [shape], dtype='uint' ))
        # set up fragment shader subroutine
        surfaceKind = inx # 0: color, 1: texture
        glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, np.array( [surfaceKind], dtype='uint'
))

        materialBuffer[inx].BindToTarget()
        glUniformMatrix4fv( modelMatLocation, 1, GL_FALSE, modelMat[inx] )
        glDrawArrays( GL_POINTS, 0, 1 )

    glutSwapBuffers()

def Fract(val): return val - math.trunc(val)
def CalcAng(currentTime, intervall): return Fract( (currentTime - startTime) / intervall ) *
360.0
def CalcMove(currentTime, intervall, range):
    pos = Fract( (currentTime - startTime) / intervall ) * 2.0
    pos = pos if pos < 1.0 else (2.0-pos)
    return range[0] + (range[1] - range[0]) * pos

# read shader program and compile shader
def CompileShader( sourceFileName, shaderStage ):
    with open( sourceFileName, 'r' ) as sourceFile:
        sourceCode = sourceFile.read()
    nameMap = { GL_VERTEX_SHADER: 'vertex', GL_GEOMETRY_SHADER: 'geometry',
GL_FRAGMENT_SHADER: 'fragment' }
    print( '\n%s shader code:' % nameMap.get(shaderStage, '') )
    print( sourceCode )
    shaderObj = glCreateShader( shaderStage )
    glShaderSource( shaderObj, sourceCode )
    glCompileShader( shaderObj )
    result = glGetShaderiv( shaderObj, GL_COMPILE_STATUS )
    if not (result):
        print( glGetShaderInfoLog( shaderObj ) )
        sys.exit()
    return shaderObj
```

```
# linke shader objects to shader program
def LinkProgram( shaderObjs ):
    shaderProgram = glCreateProgram()
    for shObj in shaderObjs:
        glAttachShader( shaderProgram, shObj )
    glLinkProgram( shaderProgram )
    result = glGetProgramiv( shaderProgram, GL_LINK_STATUS )
    if not (result):
        print( 'link error:' )
        print( glGetProgramInfoLog( shaderProgram ) )
        sys.exit()
    return shaderProgram

# create vertex array object
def CreateVAO( dataArrays ):
    noOfBuffers = len(dataArrays)
    buffers = glGenBuffers(noOfBuffers)
    newVAObj = glGenVertexArrays( 1 )
    glBindVertexArray( newVAObj )
    for inx in range(0, noOfBuffers):
        vertexSize, dataArr = dataArrays[inx]
        arr = np.array( dataArr, dtype='float32' )
        glBindBuffer( GL_ARRAY_BUFFER, buffers[inx] )
        glBufferData( GL_ARRAY_BUFFER, arr, GL_STATIC_DRAW )
        glEnableVertexAttribArray( inx )
        glVertexAttribPointer( inx, vertexSize, GL_FLOAT, GL_FALSE, 0, None )
    return newVAObj

# representation of a uniform block
class UniformBlock:
     def __init__(self, shaderProg, name):
        self.shaderProg = shaderProg
        self.name = name
     def Link(self, bindingPoint):
        self.bindingPoint = bindingPoint
        self.noOfUniforms = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORMS)
        self.maxUniformNameLen = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORM_MAX_LENGTH)
        self.index = glGetUniformBlockIndex(self.shaderProg, self.name)
        intData = np.zeros(1, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS, intData)
        self.count = intData[0]
        self.indices = np.zeros(self.count, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, self.indices)
        self.offsets = np.zeros(self.count, dtype=int)
        glGetActiveUniformsiv(self.shaderProg, self.count, self.indices, GL_UNIFORM_OFFSET,
self.offsets)
        strLengthData = np.zeros(1, dtype=int)
        arraysizeData = np.zeros(1, dtype=int)
        typeData = np.zeros(1, dtype='uint32')
        nameData = np.chararray(self.maxUniformNameLen+1)
        self.namemap = {}
        self.dataSize = 0
        for inx in range(0, len(self.indices)):
            glGetActiveUniform( self.shaderProg, self.indices[inx], self.maxUniformNameLen,
strLengthData, arraysizeData,     typeData, nameData.data )
            name = nameData.tostring()[:strLengthData[0]]
            self.namemap[name] = inx
            self.dataSize = max(self.dataSize, self.offsets[inx] + arraysizeData * 16)
        glUniformBlockBinding(self.shaderProg, self.index, self.bindingPoint)
```

```
        print('\nuniform block %s size:%4d' % (self.name, self.dataSize))
        for uName in self.namemap:
            print( '    %-40s index:%2d    offset:%4d' % (uName,
self.indices[self.namemap[uName]], self.offsets[self.namemap    [uName]]) )

# representation of a uniform block buffer
class UniformBlockBuffer:
    def __init__(self, ub):
        self.namemap = ub.namemap
        self.offsets = ub.offsets
        self.bindingPoint = ub.bindingPoint
        self.object = glGenBuffers(1)
        self.dataSize = ub.dataSize
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.zeros(self.dataSize//4, dtype='float32')
        glBufferData(GL_UNIFORM_BUFFER, self.dataSize, dataArray, GL_DYNAMIC_DRAW)
    def BindToTarget(self):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        glBindBufferBase(GL_UNIFORM_BUFFER, self.bindingPoint, self.object)
    def BindDataFloat(self, name, dataArr):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.array(dataArr, dtype='float32')
        glBufferSubData(GL_UNIFORM_BUFFER, self.offsets[self.namemap[name]], len(dataArr)*4,
dataArray)

def Translate(matA, trans):
    matB = np.copy(matA)
    for i in range(0, 4): matB[3,i] = matA[0,i] * trans[0] + matA[1,i] * trans[1] + matA[2,i]
* trans[2] + matA[3,i]
    return matB

def Scale(matA, s):
    matB = np.copy(matA)
    for i0 in range(0, 3):
        for i1 in range(0, 4): matB[i0,i1] = matA[i0,i1] * s[i0]
    return matB

def RotateHlp(matA, angDeg, a0, a1):
    matB = np.copy(matA)
    ang = math.radians(angDeg)
    sinAng, cosAng = math.sin(ang), math.cos(ang)
    for i in range(0, 4):
        matB[a0,i] = matA[a0,i] * cosAng + matA[a1,i] * sinAng
        matB[a1,i] = matA[a0,i] * -sinAng + matA[a1,i] * cosAng
    return matB

def RotateX(matA, angDeg): return RotateHlp(matA, angDeg, 1, 2)
def RotateY(matA, angDeg): return RotateHlp(matA, angDeg, 2, 0)
def RotateZ(matA, angDeg): return RotateHlp(matA, angDeg, 0, 1)
def RotateView(matA, angDeg): return RotateZ(RotateY(RotateX(matA, angDeg[0]), angDeg[1]),
angDeg[2])

def Multiply(matA, matB):
    matC = np.copy(matA)
    for i0 in range(0, 4):
        for i1 in range(0, 4):
            matC[i0,i1] = matB[i0,0] * matA[0,i1] + matB[i0,1] * matA[1,i1] + matB[i0,2] *
matA[2,i1] + matB[i0,3] * matA[3,i1]
    return matC

def ToMat33(mat44):
```

```
    mat33 = np.matrix(np.identity(3), copy=False, dtype='float32')
    for i0 in range(0, 3):
        for i1 in range(0, 3): mat33[i0, i1] = mat44[i0, i1]
    return mat33

def TransformVec4(vecA,mat44):
    vecB = np.zeros(4, dtype='float32')
    for i0 in range(0, 4):
        vecB[i0] = vecA[0] * mat44[0,i0] + vecA[1] * mat44[1,i0] + vecA[2] * mat44[2,i0]  +
vecA[3] * mat44[3,i0]
    return vecB

def Perspective(fov, aspectRatio, near, far):
    fn, f_n = far + near, far - near
    r, t = aspectRatio, 1.0 / math.tan( math.radians(fov) / 2.0 )
    return np.matrix( [ [t/r,0,0,0], [0,t,0,0], [0,0,-fn/f_n,-2.0*far*near/f_n], [0,0,-1,0] ]
)

def AddToBuffer( buffer, data, count=1 ):
    for inx_c in range(0, count):
        for inx_s in range(0, len(data)): buffer.append( data[inx_s] )

# initialize glut
glutInit()

# create window
wndW, wndH = 800, 600
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
glutInitWindowPosition(0, 0)
glutInitWindowSize(wndW, wndH)
wndID = glutCreateWindow(b'OGL window')
glutDisplayFunc(OnDraw)
glutIdleFunc(OnDraw)

# define location vertex array opject
pointVAObj = CreateVAO( [ (3, [0.0, 0.0, 0.0] ), (3, [0.0, 0.0, 1.0]), (3, [1.0, 0.0, 0.0]) ]
)

# create texture
texCX, texCY = 128, 128
texPlan = np.zeros( texCX * texCY * 4, dtype=np.uint8 )
for inx_x in range(0, texCX):
    for inx_y in range(0, texCY):
        val_x = math.sin( math.pi * 6.0 * inx_x / texCX )
        val_y = math.sin( math.pi * 6.0 * inx_y / texCY )
        inx_tex = inx_y * texCX * 4 + inx_x * 4
        texPlan[inx_tex + 0] = int( 128 + 127 * val_x )
        texPlan[inx_tex + 1] = 63
        texPlan[inx_tex + 2] = int( 128 + 127 * val_y )
        texPlan[inx_tex + 3] = 255
glActiveTexture( GL_TEXTURE0 )
texObj = glGenTextures( 1  )
glBindTexture( GL_TEXTURE_2D, texObj )
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texCX, texCY, 0, GL_RGBA, GL_UNSIGNED_BYTE, texPlan)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)

# load, compile and link shader
shaderProgram = LinkProgram( [
```

```
        CompileShader( 'python/ogl4subr/subr.vert', GL_VERTEX_SHADER ),
        CompileShader( 'python/ogl4subr/subr.geo', GL_GEOMETRY_SHADER ),
        CompileShader( 'python/ogl4subr/subr.frag', GL_FRAGMENT_SHADER )
    ] )
# get unifor locations
projectionMatLocation = glGetUniformLocation(shaderProgram, "u_projectionMat44")
viewMatLocation       = glGetUniformLocation(shaderProgram, "u_viewMat44")
modelMatLocation      = glGetUniformLocation(shaderProgram, "u_modelMat44")
textureMatLocation    = glGetUniformLocation(shaderProgram, "u_textureMat44")
textureLocation       = glGetUniformLocation(shaderProgram, "u_texture")
# linke uniform blocks
ubMaterial = UniformBlock(shaderProgram, "UB_material")
ubLightSource = UniformBlock(shaderProgram, "UB_lightSource")
ubMaterial.Link(1)
ubLightSource.Link(2)

# create uniform block buffers
lightSourceBuffer = UniformBlockBuffer(ubLightSource)
lightSourceBuffer.BindDataFloat(b'u_lightSource.ambient', [0.2, 0.2, 0.2, 1.0])
lightSourceBuffer.BindDataFloat(b'u_lightSource.diffuse', [0.2, 0.2, 0.2, 1.0])
lightSourceBuffer.BindDataFloat(b'u_lightSource.specular', [1.0, 1.0, 1.0, 1.0])

materialBuffer = [ UniformBlockBuffer(ubMaterial), UniformBlockBuffer(ubMaterial) ]

materialBuffer[0].BindDataFloat(b'u_roughness', [0.45])
materialBuffer[0].BindDataFloat(b'u_fresnel0', [0.45])
materialBuffer[0].BindDataFloat(b'u_color', [0.5, 0.7, 0.6, 1.0])
materialBuffer[0].BindDataFloat(b'u_specularTint',[1.0, 0.5, 0.5, 0.8])

materialBuffer[1].BindDataFloat(b'u_roughness', [0.4])
materialBuffer[1].BindDataFloat(b'u_fresnel0', [0.4])
materialBuffer[1].BindDataFloat(b'u_color', [0.7, 0.5, 0.6, 1.0])
materialBuffer[1].BindDataFloat(b'u_specularTint',[0.5, 1.0, 0.5, 0.8])

# start main loop
startTime = time()
glutMainLoop()
```

Changing the geometry with tessellation shaders in OGL 4.0 GLSL

A simple OGL 4.0 GLSL shader program that shows that shows how to add details with tessellation shader to the geometry. The program is executed with a python script. To run the script, PyOpenGL and NumPy must be installed.

The basic mesh in this example is an icosahedron that consists of 20 triangles. The tessellation control shader defines how each triangle is divided into a set of many small parts. When tessellating a triangle, the generated data are barycentric coordinates based on the original triangle. The tessellation evaluation shader generates new geometry from the data obtained in this way. In this example, each triangle gets a peak in the middle, which rises outward from the center of the icosader. In this way a much more complex geometry is generated than the original icosahedron.

Vertex shader

tess.vert

```
layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inNV;

out TVertexData
{
    vec3 pos;
    vec3 nv;
} outData;

uniform mat4 u_projectionMat44;
uniform mat4 u_modelViewMat44;
uniform mat3 u_normalMat33;

void main()
{
    vec4 viewPos = u_modelViewMat44 * vec4( inPos, 1.0 );

    outData.pos = viewPos.xyz / viewPos.w;
    outData.nv  = u_normalMat33 * normalize( inNV );

    gl_Position = u_projectionMat44 * viewPos;
}
```

Tessellation control shader

tess.tctrl

```
#version 400

layout( vertices=3 ) out;

in TVertexData
{
    vec3 pos;
    vec3 nv;
} inData[];

out TVertexData
{
    vec3 pos;
    vec3 nv;
} outData[];

void main()
{
    outData[gl_InvocationID].pos = inData[gl_InvocationID].pos;
    outData[gl_InvocationID].nv  = inData[gl_InvocationID].nv;

    if ( gl_InvocationID == 0 )
    {
        gl_TessLevelOuter[0] = 10.0;
        gl_TessLevelOuter[1] = 10.0;
        gl_TessLevelOuter[2] = 10.0;
        gl_TessLevelInner[0] = 10.0;
    }
}
```

Tessellation evaluation shader

tess.teval

```
#version 400

layout(triangles, equal_spacing, ccw) in;

in TVertexData
{
    vec3 pos;
    vec3 nv;
} inData[];

out TTessData
{
    vec3  pos;
    vec3  nv;
    float height;
} outData;

uniform mat4 u_projectionMat44;

void main()
{
    float sideLen[3] = float[3]
    (
        length( inData[1].pos - inData[0].pos ),
        length( inData[2].pos - inData[1].pos ),
        length( inData[0].pos - inData[2].pos )
    );
    float s = ( sideLen[0] + sideLen[1] + sideLen[2] ) / 2.0;
    float rad = sqrt( (s - sideLen[0]) * (s - sideLen[1]) * (s - sideLen[2]) / s );

    vec3 cpt = ( inData[0].pos + inData[1].pos + inData[2].pos ) / 3.0;
    vec3 pos = inData[0].pos * gl_TessCoord.x + inData[1].pos * gl_TessCoord.y + inData[2].pos
* gl_TessCoord.z;
    vec3 nv  = normalize( inData[0].nv * gl_TessCoord.x + inData[1].nv * gl_TessCoord.y +
inData[2].nv * gl_TessCoord.z );

    float cptDist      = length( cpt - pos );
    float sizeRelation = 1.0 - min( rad, cptDist ) / rad;
    float height       = pow( sizeRelation, 2.0 );

    outData.pos    = pos + nv * height * rad;
    outData.nv     = mix( nv, normalize( pos - cpt ), height );
    outData.height = height;

    gl_Position = u_projectionMat44 * vec4( outData.pos, 1.0 );
}
```

Fragment shader

tess.frag

```
#version 400

in TTessData
{
    vec3  pos;
```

```
    vec3  nv;
    float height;
} inData;

out vec4 fragColor;

uniform sampler2D u_texture;

uniform UB_material
{
    float u_roughness;
    float u_fresnel0;
    vec4  u_color;
    vec4  u_specularTint;
};

struct TLightSource
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 dir;
};

uniform UB_lightSource
{
    TLightSource u_lightSource;
};

float Fresnel_Schlick( in float theta );
vec3 LightModel( in vec3 esPt, in vec3 esPtNV, in vec3 col, in vec4 specularTint, in float
roughness, in float fresnel0 );

void main()
{
    vec3 col = mix( u_color.rgb, vec3( 1.0, 1.0, 1.0 ), inData.height );
    vec3 lightCol = LightModel( inData.pos, inData.nv, col, u_specularTint, u_roughness,
u_fresnel0 );
    fragColor = vec4( clamp( lightCol, 0.0, 1.0 ), 1.0 );
}

float Fresnel_Schlick( in float theta )
{
    float m = clamp( 1.0 - theta, 0.0, 1.0 );
    float m2 = m * m;
    return m2 * m2 * m; // pow( m, 5.0 )
}

vec3 LightModel( in vec3 esPt, in vec3 esPtNV, in vec3 col, in vec4 specularTint, in float
roughness, in float fresnel0 )
{
  vec3  esVLight      = normalize( -u_lightSource.dir.xyz );
  vec3  esVEye        = normalize( -esPt );
  vec3  halfVector    = normalize( esVEye + esVLight );
  float HdotL         = dot( halfVector, esVLight );
  float NdotL         = dot( esPtNV, esVLight );
  float NdotV         = dot( esPtNV, esVEye );
  float NdotH         = dot( esPtNV, halfVector );
  float NdotH2        = NdotH * NdotH;
  float NdotL_clamped = max( NdotL, 0.0 );
  float NdotV_clamped = max( NdotV, 0.0 );
```

```
  float m2            = roughness * roughness;

  // Lambertian diffuse
  float k_diffuse = NdotL_clamped;
  // Schlick approximation
  float fresnel = fresnel0 + ( 1.0 - fresnel0 ) * Fresnel_Schlick( HdotL );
  // Beckmann distribution
  float distribution = max( 0.0, exp( ( NdotH2 - 1.0 ) / ( m2 * NdotH2 ) ) / ( 3.14159265 * m2
* NdotH2 * NdotH2 ) );
  // Torrance-Sparrow geometric term
  float geometric_att = min( 1.0, min( 2.0 * NdotH * NdotV_clamped / HdotL, 2.0 * NdotH *
NdotL_clamped / HdotL ) );
  // Microfacet bidirectional reflectance distribution function
  float k_specular = fresnel * distribution * geometric_att / ( 4.0 * NdotL_clamped *
NdotV_clamped );

  vec3 lightColor = col.rgb * u_lightSource.ambient.rgb +
                    max( 0.0, k_diffuse ) * col.rgb * u_lightSource.diffuse.rgb +
                    max( 0.0, k_specular ) * mix( col.rgb, specularTint.rgb, specularTint.a )
* u_lightSource.specular.rgb;
  return lightColor;
}
```

Python script

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
from time import time
import math
import sys

sin120 = 0.8660254
rotateCamera = False

# draw event
def OnDraw():
    dist = 3.0
    currentTime = time()
    comeraRotAng = CalcAng( currentTime, 10.0 )
    # set up projection matrix
    prjMat = Perspective(90.0, wndW/wndH, 0.5, 100.0)
    # set up view matrix
    viewMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    viewMat = Translate( viewMat, np.array( [0.0, 0.0, -12.0] ) )
    viewMat = RotateView( viewMat, [30.0, comeraRotAng if rotateCamera else 0.0, 0.0] )

    # set up light source
    lightSourceBuffer.BindDataFloat(b'u_lightSource.dir', TransformVec4([-1.0, -1.0, -5.0,
0.0], viewMat) )

    # set up icosahedron model matrix
    icoModelMat = np.matrix(np.identity(4), copy=False, dtype='float32')
    if not rotateCamera: icoModelMat = RotateY( icoModelMat, comeraRotAng )
    icoModelMat = Scale( icoModelMat, np.repeat( 5, 3 ) )
    icoModelMat = RotateY( icoModelMat, CalcAng( currentTime, 17.0 ) )
    icoModelMat = RotateX( icoModelMat, CalcAng( currentTime, 13.0 ) )
```

```
    # set up attributes and shader program
    glEnable( GL_DEPTH_TEST )
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
    glUseProgram( shaderProgram )
    glUniformMatrix4fv( projectionMatLocation, 1, GL_FALSE, prjMat )
    lightSourceBuffer.BindToTarget()

    # draw icosahedron
    icoMaterialBuffer.BindToTarget()
    modelViewMat = Multiply(viewMat, icoModelMat)
    glUniformMatrix4fv( modelViewMatLocation, 1, GL_FALSE, modelViewMat )
    glUniformMatrix3fv( normalMatLocation, 1, GL_FALSE, ToMat33(modelViewMat) )
    glBindVertexArray( icoVAObj )
    glPatchParameteri( GL_PATCH_VERTICES, 3 )
    glDrawArrays( GL_PATCHES, 0, len(icoPosData) )

    glutSwapBuffers()

def Fract(val): return val - math.trunc(val)
def CalcAng(currentTime, intervall): return Fract( (currentTime - startTime) / intervall ) *
360.0
def CalcMove(currentTime, intervall, range):
    pos = Fract( (currentTime - startTime) / intervall ) * 2.0
    pos = pos if pos < 1.0 else (2.0-pos)
    return range[0] + (range[1] - range[0]) * pos

# read shader program and compile shader
def CompileShader( sourceFileName, shaderStage ):
    with open( sourceFileName, 'r' ) as sourceFile:
        sourceCode = sourceFile.read()
    nameMap = { GL_VERTEX_SHADER: 'vertex', GL_GEOMETRY_SHADER: 'geometry',
GL_FRAGMENT_SHADER: 'fragment' }
    print( '\n%s shader code:' % nameMap.get(shaderStage, '') )
    print( sourceCode )
    shaderObj = glCreateShader( shaderStage )
    glShaderSource( shaderObj, sourceCode )
    glCompileShader( shaderObj )
    result = glGetShaderiv( shaderObj, GL_COMPILE_STATUS )
    if not (result):
        print( glGetShaderInfoLog( shaderObj ) )
        sys.exit()
    return shaderObj

# link shader objects to shader program
def LinkProgram( shaderObjs ):
    shaderProgram = glCreateProgram()
    for shObj in shaderObjs:
        glAttachShader( shaderProgram, shObj )
    glLinkProgram( shaderProgram )
    result = glGetProgramiv( shaderProgram, GL_LINK_STATUS )
    if not (result):
        print( 'link error:' )
        print( glGetProgramInfoLog( shaderProgram ) )
        sys.exit()
    return shaderProgram

# create vertex array object
def CreateVAO( dataArrays ):
    noOfBuffers = len(dataArrays)
    buffers = glGenBuffers(noOfBuffers)
    newVAObj = glGenVertexArrays( 1 )
```

```
    glBindVertexArray( newVAObj )
    for inx in range(0, noOfBuffers):
        vertexSize, dataArr = dataArrays[inx]
        arr = np.array( dataArr, dtype='float32' )
        glBindBuffer( GL_ARRAY_BUFFER, buffers[inx] )
        glBufferData( GL_ARRAY_BUFFER, arr, GL_STATIC_DRAW )
        glEnableVertexAttribArray( inx )
        glVertexAttribPointer( inx, vertexSize, GL_FLOAT, GL_FALSE, 0, None )
    return newVAObj

# representation of a uniform block
class UniformBlock:
     def __init__(self, shaderProg, name):
        self.shaderProg = shaderProg
        self.name = name
     def Link(self, bindingPoint):
        self.bindingPoint = bindingPoint
        self.noOfUniforms = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORMS)
        self.maxUniformNameLen = glGetProgramiv(self.shaderProg, GL_ACTIVE_UNIFORM_MAX_LENGTH)
        self.index = glGetUniformBlockIndex(self.shaderProg, self.name)
        intData = np.zeros(1, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS, intData)
        self.count = intData[0]
        self.indices = np.zeros(self.count, dtype=int)
        glGetActiveUniformBlockiv(self.shaderProg, self.index,
GL_UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, self.indices)
        self.offsets = np.zeros(self.count, dtype=int)
        glGetActiveUniformsiv(self.shaderProg, self.count, self.indices, GL_UNIFORM_OFFSET,
self.offsets)
        strLengthData = np.zeros(1, dtype=int)
        arraysizeData = np.zeros(1, dtype=int)
        typeData = np.zeros(1, dtype='uint32')
        nameData = np.chararray(self.maxUniformNameLen+1)
        self.namemap = {}
        self.dataSize = 0
        for inx in range(0, len(self.indices)):
            glGetActiveUniform( self.shaderProg, self.indices[inx], self.maxUniformNameLen,
strLengthData, arraysizeData,     typeData, nameData.data )
            name = nameData.tostring()[:strLengthData[0]]
            self.namemap[name] = inx
            self.dataSize = max(self.dataSize, self.offsets[inx] + arraysizeData * 16)
        glUniformBlockBinding(self.shaderProg, self.index, self.bindingPoint)
        print('\nuniform block %s size:%4d' % (self.name, self.dataSize))
        for uName in self.namemap:
            print( '    %-40s index:%2d    offset:%4d' % (uName,
self.indices[self.namemap[uName]], self.offsets[self.namemap    [uName]]) )

# representation of a uniform block buffer
class UniformBlockBuffer:
    def __init__(self, ub):
        self.namemap = ub.namemap
        self.offsets = ub.offsets
        self.bindingPoint = ub.bindingPoint
        self.object = glGenBuffers(1)
        self.dataSize = ub.dataSize
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.zeros(self.dataSize//4, dtype='float32')
        glBufferData(GL_UNIFORM_BUFFER, self.dataSize, dataArray, GL_DYNAMIC_DRAW)
    def BindToTarget(self):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
```

```
        glBindBufferBase(GL_UNIFORM_BUFFER, self.bindingPoint, self.object)
    def BindDataFloat(self, name, dataArr):
        glBindBuffer(GL_UNIFORM_BUFFER, self.object)
        dataArray = np.array(dataArr, dtype='float32')
        glBufferSubData(GL_UNIFORM_BUFFER, self.offsets[self.namemap[name]], len(dataArr)*4,
dataArray)

def Translate(matA, trans):
    matB = np.copy(matA)
    for i in range(0, 4): matB[3,i] = matA[0,i] * trans[0] + matA[1,i] * trans[1] + matA[2,i]
* trans[2] + matA[3,i]
    return matB

def Scale(matA, s):
    matB = np.copy(matA)
    for i0 in range(0, 3):
        for i1 in range(0, 4): matB[i0,i1] = matA[i0,i1] * s[i0]
    return matB

def RotateHlp(matA, angDeg, a0, a1):
    matB = np.copy(matA)
    ang = math.radians(angDeg)
    sinAng, cosAng = math.sin(ang), math.cos(ang)
    for i in range(0, 4):
        matB[a0,i] = matA[a0,i] * cosAng + matA[a1,i] * sinAng
        matB[a1,i] = matA[a0,i] * -sinAng + matA[a1,i] * cosAng
    return matB

def RotateX(matA, angDeg): return RotateHlp(matA, angDeg, 1, 2)
def RotateY(matA, angDeg): return RotateHlp(matA, angDeg, 2, 0)
def RotateZ(matA, angDeg): return RotateHlp(matA, angDeg, 0, 1)
def RotateView(matA, angDeg): return RotateZ(RotateY(RotateX(matA, angDeg[0]), angDeg[1]),
angDeg[2])

def Multiply(matA, matB):
    matC = np.copy(matA)
    for i0 in range(0, 4):
        for i1 in range(0, 4):
            matC[i0,i1] = matB[i0,0] * matA[0,i1] + matB[i0,1] * matA[1,i1] + matB[i0,2] *
matA[2,i1] + matB[i0,3] * matA[3,i1]
    return matC

def ToMat33(mat44):
    mat33 = np.matrix(np.identity(3), copy=False, dtype='float32')
    for i0 in range(0, 3):
        for i1 in range(0, 3): mat33[i0, i1] = mat44[i0, i1]
    return mat33

def TransformVec4(vecA,mat44):
    vecB = np.zeros(4, dtype='float32')
    for i0 in range(0, 4):
        vecB[i0] = vecA[0] * mat44[0,i0] + vecA[1] * mat44[1,i0] + vecA[2] * mat44[2,i0]  +
vecA[3] * mat44[3,i0]
    return vecB

def Perspective(fov, aspectRatio, near, far):
    fn, f_n = far + near, far - near
    r, t = aspectRatio, 1.0 / math.tan( math.radians(fov) / 2.0 )
    return np.matrix( [ [t/r,0,0,0], [0,t,0,0], [0,0,-fn/f_n,-2.0*far*near/f_n], [0,0,-1,0] ]
)
```

```
def AddToBuffer( buffer, data, count=1 ):
    for inx_c in range(0, count):
        for inx_s in range(0, len(data)): buffer.append( data[inx_s] )

# initialize glut
glutInit()

# create window
wndW, wndH = 800, 600
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
glutInitWindowPosition(0, 0)
glutInitWindowSize(wndW, wndH)
wndID = glutCreateWindow(b'OGL window')
glutDisplayFunc(OnDraw)
glutIdleFunc(OnDraw)

# define icosahedron vertex array opject
icoPts = [
    ( 0.000,  0.000,  1.000), ( 0.894,  0.000,  0.447), ( 0.276,  0.851,  0.447), (-0.724,
0.526,  0.447),
    (-0.724, -0.526,  0.447), ( 0.276, -0.851,  0.447), ( 0.724,  0.526, -0.447), (-0.276,
0.851, -0.447),
    (-0.894,  0.000, -0.447), (-0.276, -0.851, -0.447), ( 0.724, -0.526, -0.447), ( 0.000,
0.000, -1.000) ]
icoCol = [ [1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [1.0, 1.0, 0.0], [0.0, 1.0, 0.0], [1.0, 0.5,
0.0], [1.0, 0.0, 1.0] ]
icoIndices = [
  2,  0,  1,  3,  0,  2,  4,  0,  3,  5,  0,  4,  1,  0,  5, 11,  7,  6, 11,  8,  7, 11,  9,
8, 11, 10,  9, 11,  6, 10,
  1,  6,  2,  2,  7,  3,  3,  8,  4,  4,  9,  5,  5, 10,  1,  2,  6,  7,  3,  7,  8,  4,  8,
9,  5,  9, 10,  1, 10,  6  ]
icoPosData = []
for inx in icoIndices: AddToBuffer( icoPosData, icoPts[inx] )
icoNVData = []
for inx_nv in range(0, len(icoIndices) // 3):
    nv = [0.0, 0.0, 0.0]
    for inx_p in range(0, 3):
        for inx_s in range(0, 3): nv[inx_s] += icoPts[ icoIndices[inx_nv*3 + inx_p] ][inx_s]
    AddToBuffer( icoNVData, nv, 3 )
icoVAObj = CreateVAO( [ (3, icoPosData), (3, icoNVData) ] )

# load, compile and link shader
shaderProgram = LinkProgram( [
        CompileShader( 'tess.vert', GL_VERTEX_SHADER ),
        CompileShader( 'tess.tctrl', GL_TESS_CONTROL_SHADER ),
        CompileShader( 'tess.teval', GL_TESS_EVALUATION_SHADER ),
        CompileShader( 'tess.frag', GL_FRAGMENT_SHADER )
    ] )
# get unifor locations
projectionMatLocation = glGetUniformLocation(shaderProgram, "u_projectionMat44")
modelViewMatLocation  = glGetUniformLocation(shaderProgram, "u_modelViewMat44")
normalMatLocation     = glGetUniformLocation(shaderProgram, "u_normalMat33")
# linke uniform blocks
ubMaterial = UniformBlock(shaderProgram, "UB_material")
ubLightSource = UniformBlock(shaderProgram, "UB_lightSource")
ubMaterial.Link(1)
ubLightSource.Link(2)

# create uniform block buffers
lightSourceBuffer = UniformBlockBuffer(ubLightSource)
lightSourceBuffer.BindDataFloat(b'u_lightSource.ambient', [0.2, 0.2, 0.2, 1.0])
```

```
lightSourceBuffer.BindDataFloat(b'u_lightSource.diffuse', [0.2, 0.2, 0.2, 1.0])
lightSourceBuffer.BindDataFloat(b'u_lightSource.specular', [1.0, 1.0, 1.0, 1.0])

icoMaterialBuffer = UniformBlockBuffer(ubMaterial)
icoMaterialBuffer.BindDataFloat(b'u_roughness', [0.45])
icoMaterialBuffer.BindDataFloat(b'u_fresnel0', [0.4])
icoMaterialBuffer.BindDataFloat(b'u_color', [0.6, 0.5, 0.8, 1.0])
icoMaterialBuffer.BindDataFloat(b'u_specularTint',[1.0, 0.5, 0.5, 0.8])

# start main loop
startTime = time()
glutMainLoop()
```

Read Getting started with glsl online: https://riptutorial.com/glsl/topic/5480/getting-started-with-glsl

Credits

S. No	Chapters	Contributors
1	Getting started with glsl	Community, KevinO, MarGenDo, Rabbid76