Цель работы.

Разработка визуализатора заданной трехмерной сцены на базе открытой графической библиотеки (OpenGL).

Основные теоретические положения.

В качестве языка-платформы для построения сцены на основе OpenGL был выбран Python и библиотека PyOpenGL. Был использован режим совместимости (версия GLSL 1.20.8) и GLUT в качестве библиотеки операций ввода-вывода. Режим совместимости используется потому, что кто-то очень любит стеки матриц и функции по преобразованию матриц вида, модели и проекции, а также не любит альтернативную передачу предопределенных в шейдерах версии 1.20.8 переменных (gl_Normal, gl_ModelViewMatrix etc.).

Прочие технические сравнения версий можно найти здесь http://retokoradi.com/2014/03/30/opengl-transition-to-core-profile/ .

Реализация.

1. **Импорт моделей в среду**

Координаты моделей были приведены к общим координатам перед импортом в OpenGL с помощью MeshLab. Прототип импортера был взят с

http://www.pygame.org/wiki/OBJFileLoader, где использовалась библиотека pygame.

Структура obj-файла следующая:

vn -0.318157 1.435133 0.000000

v 58.480572 40.125370 13.802120

vn -0.459765 2.073826 -0.000005

v 63.786060 41.301571 51.403149

…

f 274//274 6//6 263//263

, где vn – нормаль вершины, v – вершина, f – поверхность-треугольник из вершин.

В импортере используется list-семантика OpenGL 2.x (glNewList, glEndList). Последовательно для каждой поверхности, перечисленной в obj-файле строится поверхность в терминах OpenGL (файл loader.py).

2. Модель освещения

Использовалась модель Фонга, реализованная на уровне фрагментного шейдера.

Код вершинного шейдера:

```
def phongVertex():
    return """
varying vec3 N;
varying vec3 v;
void main(void)
{
   v = vec3(gl_ModelViewMatrix * gl_Vertex);
   N = normalize(gl_NormalMatrix * gl_Normal);
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}"""
```

На данном этапе высчитываются координаты вершины в системе координат, связанной с камерой (v), нормаль, нормализованная к единичной (N), а также координата на экране (gl_Position).

Код фрагментного шейдера:

```
def phongFragment():
    return """
varying vec3 N;
varying vec3 v;
```

```
void main (void)
{
   vec3 L = normalize(gl_LightSource[0].position.xyz - v);
   vec3 E = normalize(-v); // we are in Eye Coordinates, so EyePos is (0,0,0)
   vec3 R = normalize(-reflect(L,N));

   //calculate Ambient Term:
   vec4 Iamb = gl_FrontLightProduct[0].ambient;

   //calculate Diffuse Term:
   vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
   Idiff = clamp(Idiff, 0.0, 1.0);

   // calculate Specular Term:
   vec4 Ispec = gl_FrontLightProduct[0].specular
                * pow(max(dot(R,E),0.0),0.3*gl_FrontMaterial.shininess);
   Ispec = clamp(Ispec, 0.0, 1.0);
   // write Total Color:
   gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
}"""
```

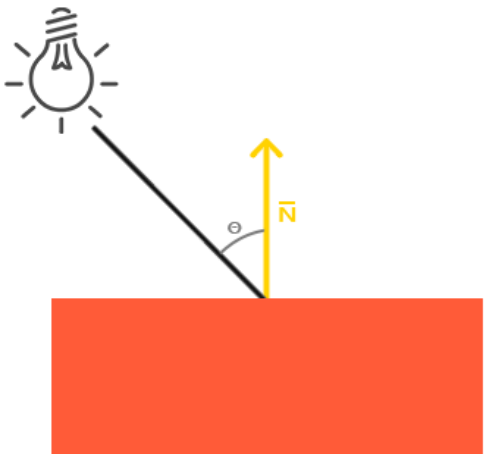
Прежде всего, модель освещения объекта складывается из следующих компонент: фоновое, диффузное, зеркальное.

Фоновое освещение (`vec4 Iamb`) не зависит ни от углов нормали фрагмента к камере, ни от нормали фрагмента к источнику света, а зависит лишь от источника света и материала:

```
gl_FrontLightProduct[0] == gl_FrontMaterial * gl_LightSource[0]
```

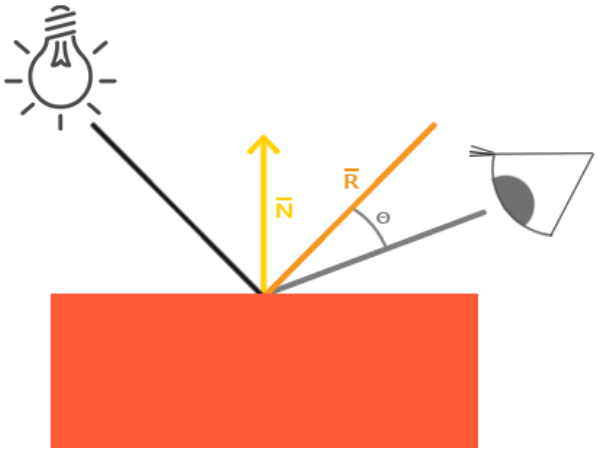
Диффузное освещение зависит от угла между нормалью к фрагменту и вектором, задающим направление от фрагмента к источнику освещения:

```
vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
```



Чем меньше угол, тем более близкое к (1,1,1) значение получится.

Зеркальное освещение зависит уже и от положения камеры к фрагменту. Точнее, от угла между отраженным относительно нормали к фрагменту вектором освещения и вектором-камеры к фрагменту:



Финальный цвет фрагмента есть взвешенная сумма всех трех типов освещения:

```
gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
```

Материал по освещению использован со след. источников:

http://learnopengl.com/#!Lighting/Basic-Lighting (реализация для OpenGL 4)

http://www.falloutsoftware.com/tutorials/gl/gl8.htm (основы освещения).

Note. В режиме поддержки совместимости в OpenGL в fixed pipeline используется т.н. Gouraud shading (когда освещение рассчитывается на каждую вершину, а не пиксель), что дает худший в сравнении результат по качеству.

3. Модель теней

Для реализации теней была использована модель Shadow mapping. За основу взят материал и код на C++ http://www.paulsprojects.net/tutorials/smt/smt.html, адаптированный под python (shadows.py). Схема метода такова:

1. Создаем текстуру для маппинга
2. Сначала рендерим сцену лишь в буфер глубины с точки зрения освещения, используя позицию камеры, равной позиции освещения, запоминаем обратную к матрице текущего афинного преобразования
3. Копируем отрендеренное в текстуру
4. Рендерим всю сцену с шейдерами из п.2
5. Используем расширение ARB (GL_TEXTURE_COMPARE_MODE_ARB) и нашу текстуру, чтобы расширение автоматически протестировало глубину точки с глубиной,

полученной на текстуре в системе координат, в которой камера расположена в нуле (это сделано), это можно сделать с помощью запомненной матрицы преобразования, и записываем результат сравнения в альфа канал.

6. Рендерим все в настроенном в п.5 окружении, используя диффузное освещение – на этом этапе будут отрисованы лишь затененные участки. Выключаем использованный в п.4 шейдер освещения, поскольку здесь (для отрисовки теней) достаточно и дефолтного для режима совместимости освещения.
7. Отключаем использованные в п.5 настройки

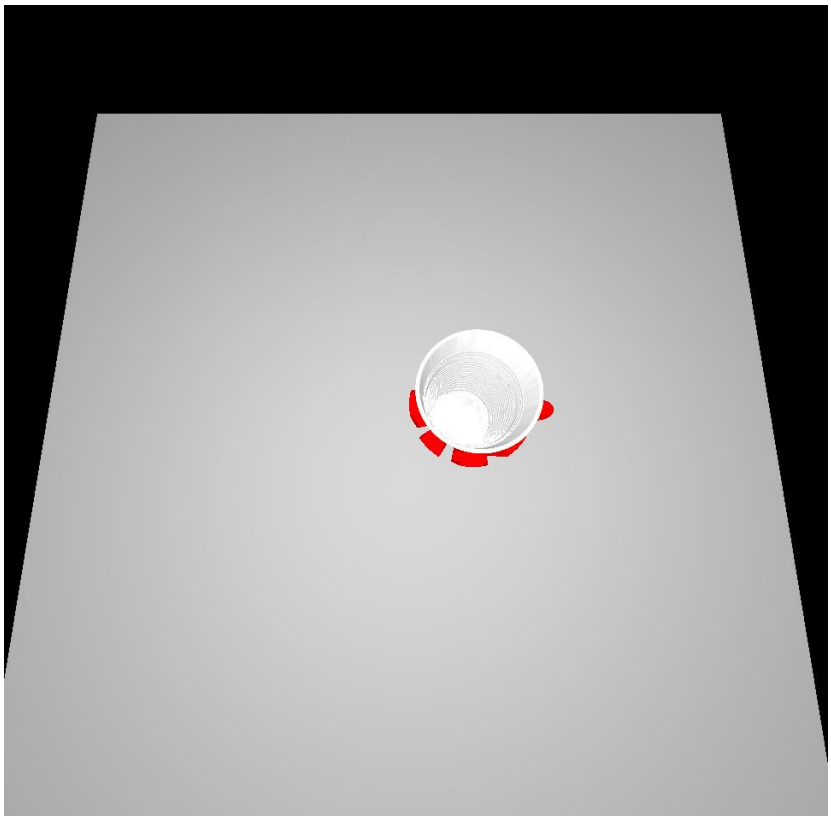


Рисунок 1. Вид на объекст с позиции освещения (пункт 2)

Данный подход был сравнен с подходом, применяемым в современном OpenGL 4.x (http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping). Плюсом использованного в работе подхода с помощью ARB является меньший объем кода, минусом по сравнению с реализацией в шейдере – третий дополнительный проход для отрисовки только тени. Еще один минус – нельзя кастомизировать полученные тени (например, сгладить края).

Основная процедура перерисовки с тенями выглядит след. образом (в соотв-и со схемой выше):

```
# Процедура перерисовки
def drawFrame():
    global lightpos
    global program

    # first pass, compute texture
    textureMatrix = CreateShadowBefore(position=lightpos)
    drawModels()
    CreateShadowAfter(textureMapID)
    # draw all objects
    cameraLoop()
    drawLighting()
    glUseProgram(program)
    drawModels()
    glUseProgram(0)
    # render only obj(s) where shadows cast
    RenderShadowCompareBefore(textureMapID, textureMatrix)
    drawModels()
    RenderShadowCompareAfter()
    glutSwapBuffers()
```

4. Прочие возможности визуализатора

Реализована возможность поворота камеры вокруг объекта с помощью стрелок:

```
def cameraLoop():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-20.0, 120.0, -20.0, 120.0, -250., 250.)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    pickObjX, pickObjY, pickObjZ = (0, 0, -1)
    radius = 100.
    eyeX = radius * cos(xrot / 180.) * sin(yrot / 180.)
    eyeY = radius * sin(xrot / 180.) * sin(yrot / 180.)
    eyeZ = radius * cos(yrot / 180.)
    print xrot,yrot
    gluLookAt(eyeX, eyeY, eyeZ, pickObjX, pickObjY, pickObjZ, 0, 1, 0)
    modelMatrix = glGetFloatv(GL_MODELVIEW_MATRIX)
    print modelMatrix
```

Построенная модель

Итоговая модель отражена на рисунках 2 и 3. Запуск главного скрипта cup.py осушествляется с помощью python cup.py .

Рисунок 2. Построенный объект

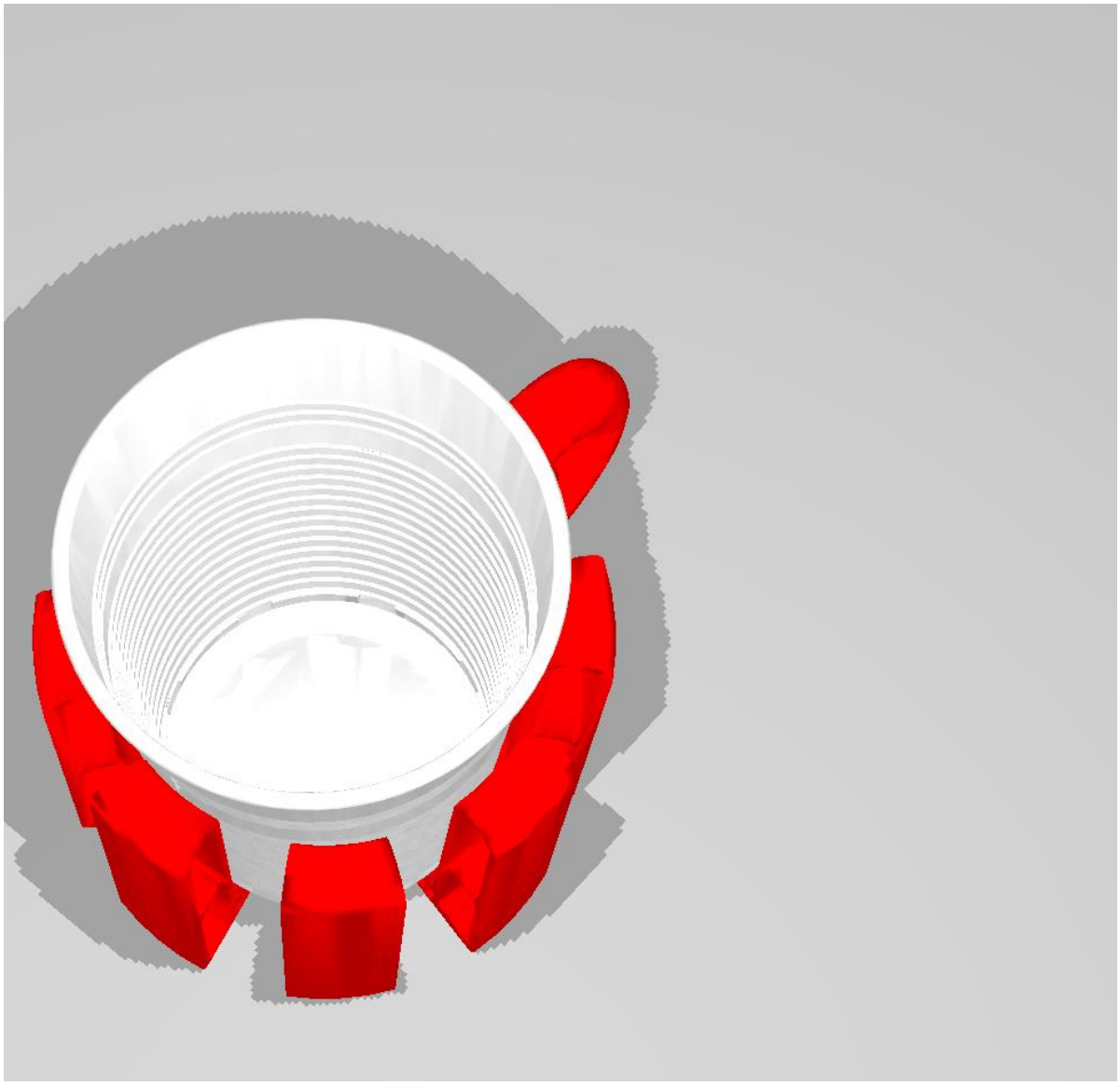


Рисунок 3. Построенный объект

Выводы

В данной работе была построена модель стакана с подстаканником в среде OpenGL. Использование конкретных версий тех или иных версий компонент графической библиотеки было обосновано. Также приведены альтернативные реализации в более “современном” подходе к задачам. В дальнейшем сложность реализации в OpenGL будет сравнена с реализациями в других графических средах.