Exceptions, homogeneous coordinates, texture coordinates, triangular faces, triangulated meshes, colors/intensities, lights and materials

– implementation details –

– a CAGD approach based on OpenGL and C++ –

Ágoston Róth

Department of Mathematics and Computer Science, Babeș-Bolyai University, Cluj-Napoca, Romania

(agoston.roth@gmail.com)

Lecture 4 – March 21, 2022

Exception handling

Exceptions.h

```cpp
#pragma once

#include <iostream>
#include <string>

namespace cagd
{
    class Exception
    {
        friend std::ostream& operator <<(std::ostream& lhs, const Exception& rhs);

    protected:
        std::string _reason;

    public:
        Exception(const std::string &reason): _reason(reason)
        {
        }
    };

    inline std::ostream& operator <<(std::ostream& lhs, const Exception& rhs)
    {
        return lhs << rhs._reason;
    }
}

try
{
    ...
    if (...)
        throw Exception("...␣reason␣...");
    ...
}
catch (Exception &e)
{
    cerr << e;
}
```

Homogeneous coordinates

Class HCoordinate3

Description

- Homogeneous coordinates utilize a mathematical trick to embed three-dimensional coordinates and transformations into a four-dimensional square matrix format. As a result, inversions or combinations of linear transformations are simplified to inversion or multiplication of the corresponding matrices. Homogeneous coordinates also make it possible to define perspective transformations.
- Note, in order to finish the implementation of the class HCoordinate3 you will need to handle points in infinity. In such cases you will be not able neither to convert homogeneous coordinates to Cartesian coordinates, nor to perform arithmetic operations.
- If $[x : y : z : w]$ is a homogeneous coordinate and the value of w is not 0, then you can convert it to a Cartesian coordinate, the components of which are x/w, y/w and z/w. If w equals to 0, then you cannot perform this conversion (such homogeneous coordinates represent points in infinity).
- If (x, y, z) is a Cartesian coordinate, you can convert it to a homogeneous one using the simple formula $[x : y : z : 1]$.

Homogeneous coordinates, part I

HCoordinates3.h, $\not\#$HCoordinates3.cpp

```cpp
#pragma once

#include <cmath>
#include <GL/glew.h>
#include <iostream>

namespace cagd
{
    //----------------------------------------
    // 3-dimensional homogeneous coordinates
    //----------------------------------------
    class HCoordinate3
    {
    protected:
        GLfloat _data[4]; // [x : y : z : w]

    public:

        // default constructor
        HCoordinate3()
        {
            _data[0] = _data[1] = _data[2] = 0.0;
            _data[3] = 1.0;
        }

        // special constructor
        HCoordinate3(GLfloat x, GLfloat y, GLfloat z = 0.0, GLfloat w = 1.0)
        {
            _data[0] = x;
            _data[1] = y;
            _data[2] = z;
            _data[3] = w;
        }
```

Homogeneous coordinates, part II

HCoordinates3.h, ∄HCoordinates3.cpp

```cpp
        // set/get
        GLfloat operator[](GLuint rhs) const;
        GLfloat x() const;
        GLfloat y() const;
        GLfloat z() const;
        GLfloat w() const;

        GLfloat& operator[](GLuint rhs);
        GLfloat& x();
        GLfloat& y();
        GLfloat& z();
        GLfloat& w();

        // add
        const HCoordinate3 operator +(const HCoordinate3& rhs) const
        {
            return HCoordinate3(
                    rhs.w() * x() + w() * rhs.x(),
                    rhs.w() * y() + w() * rhs.y(),
                    rhs.w() * z() + w() * rhs.z(),
                    w() * rhs.w());
        }

        // add to this
        HCoordinate3& operator +=(const HCoordinate3& rhs);

        // subtract
        const HCoordinate3 operator -(const HCoordinate3& rhs) const;

        // subtract from this
        HCoordinate3& operator -=(const HCoordinate3& rhs);

        // dot product
        const GLfloat operator *(const HCoordinate3& rhs) const;
```

Homogeneous coordinates, part III

HCoordinates3.h, ∄HCoordinates3.cpp

```
        // cross product
        const HCoordinate3 operator ^(const HCoordinate3& rhs) const;

        // cross product with this
        HCoordinate3& operator ^=(const HCoordinate3& rhs);

        // multiplicate with scalar from right
        const HCoordinate3 operator *(GLfloat rhs) const;

        // multiplicate this with a scalar
        HCoordinate3& operator *=(GLfloat rhs);

        // divide with scalar
        const HCoordinate3 operator /(GLfloat rhs) const;

        // divide this with a scalar
        HCoordinate3& operator /=(GLfloat rhs);

        const GLfloat length() const;

        HCoordinate3& normalize();
    };

    // scale from left with a scalar
    inline const HCoordinate3 operator *(GLfloat lhs, const HCoordinate3& rhs);

    // output to stream
    inline std::ostream& operator <<(std::ostream& lhs, const HCoordinate3& rhs);

    // input from stream
    inline std::istream& operator >>(std::istream& lhs, HCoordinate3& rhs);
}
```

Texture coordinates

Class TCoordinate4

Description

- OpenGL uses the concept of texture coordinates to achieve texture mapping. These are stored per-vertex and are interpolated in areas where there are no vertices.
- Texture coordinates can comprise one, two, three, or four coordinates. They are usually referred to as s-,t-,r-,q-coordinates to distinguish them from object coordinates (x, y,z and w) and from evaluator coordinates (u and v).
- The q-coordinate, like w, can be used to create homogeneous coordinates.
- In general, we will use 2-dimensional texture images, thus, in most of the cases a texture coordinate can be described as a pair $(s, t) \in [0, 1] \times [0, 1]$. Coordinates s and t can be greater than 1 (e.g. when we intend to repeat/clamp textures).
- **Homework**: study Chapter 9 (Texture mapping) of the OpenGL Programming Guide.

Texture coordinates – header file, part I

TCoordinates4.h, ∄ TCoordinates4.cpp

```
#pragma once

#include <GL/glew.h>
#include <iostream>

namespace cagd
{
    //----------------------------------------------------------
    // four dimensional texture coordinates
    //----------------------------------------------------------
    class TCoordinate4
    {
    protected:
        GLfloat _data[4]; // (s, t, r, q)

    public:

        // default constructor
        TCoordinate4()
        {
            _data[0] = _data[1] = _data[2] = 0.0;
            _data[3] = 1.0;
        }

        // special constructor
        TCoordinate4(GLfloat s, GLfloat t, GLfloat r = 0.0, GLfloat q = 1.0);

        // get components by value
        GLfloat operator[](GLuint rhs) const;
        GLfloat s() const;
        GLfloat t() const;
        GLfloat r() const;
        GLfloat q() const;
```

Texture coordinates – header file, part II

TCoordinates4.h, ∄ TCoordinates4.cpp

```cpp
        // get components by reference
        GLfloat& operator[](GLuint rhs);
        GLfloat& s();
        GLfloat& t();
        GLfloat& r();
        GLfloat& q();
    };

    // homework: output/input to/from stream
    inline std::ostream& operator <<(std::ostream& lhs, const TCoordinate4& rhs)
    {
        return lhs << rhs.s() << "␣" << rhs.t() << rhs.r() << "␣" << rhs.q();
    }

    inline std::istream& operator >>(std::istream& lhs, TCoordinate4& rhs);
}
```

Triangular faces

Class TriangularFace

Description

- Class TriangularFace stores three vertex indices that determine a face of our models, parametric surfaces, tensor product surfaces, etc.
- For efficiency reasons we build geometry out of triangles, since we do not have to worry about planarity or convexity of rendering primitives.

Triangular faces – header file, part I

TriangularFaces.h, ∄ TriangularFaces.cpp

```cpp
#pragma once

#include <GL/glew.h>
#include <iostream>
#include <vector>

namespace cagd
{
    class TriangularFace
    {
        // output to stream
        friend std::ostream& operator <<(std::ostream& lhs, const TriangularFace& rhs);

        // input from stream
        friend std::istream& operator >>(std::istream& lhs, TriangularFace& rhs);

    protected:
        GLuint _node[3];

    public:
        // default constructor
        TriangularFace()
        {
            _node[0] = _node[1] = _node[2] = 0;
        }

        // homework: copy constructor
        TriangularFace(const TriangularFace& face);

        // homework: assignment operator
        TriangularFace& operator =(const TriangularFace& rhs);

        // homework: get node identifiers by value
        GLuint operator [](GLuint i) const;
```

Triangular faces – header file, part II

TriangularFaces.h, ∄ TriangularFaces.cpp

```cpp
        // homework: get node identifiers by reference
        GLuint& operator [](GLuint i);
    };

    // output to stream
    inline std::ostream& operator <<(std::ostream& lhs, const TriangularFace& rhs)
    {
        lhs << 3;
        for (GLuint i = 0; i < 3; ++i)
            lhs << " " << rhs[i];
        return lhs;
    }

    // homework
    inline std::istream& operator >>(std::istream& lhs, TriangularFace& rhs);
}
```

Triangulated meshes

Class TriangulatedMesh3

Description

- For efficiency reasons we build geometry out of triangles, since we do not have to worry about planarity or convexity of rendering primitives.

Triangulated meshes – header file, part I

TriangulatedMeshes3.h

```
#pragma once

#include "DCoordinates3.h"
#include <GL/glew.h>
#include <iostream>
#include <string>
#include "TriangularFaces.h"
#include "TCoordinates4.h"
#include <vector>

namespace cagd
{
    class TriangulatedMesh3
    {
        friend class TensorProductSurface3;

        // homework: output to stream:
        // vertex count, face count
        // list of vertices
        // list of unit normal vectors
        // list of texture coordinates
        // list of faces
        friend std::ostream& operator <<(std::ostream& lhs, const TriangulatedMesh3& rhs);

        // homework: input from stream: inverse of the ostream operator
        friend std::istream& operator >>(std::istream& lhs, TriangulatedMesh3& rhs);

    protected:
        // vertex buffer object identifiers
        GLenum                      _usage_flag;
        GLuint                      _vbo_vertices;
        GLuint                      _vbo_normals;
        GLuint                      _vbo_tex_coordinates;
        GLuint                      _vbo_indices;

```

Triangulated meshes – header file, part II

TriangulatedMeshes3.h

```
        // corners of the bounding box
        DCoordinate3                  _leftmost_vertex;
        DCoordinate3                  _rightmost_vertex;

        // geometry
        std::vector<DCoordinate3>     _vertex;
        std::vector<DCoordinate3>     _normal;
        std::vector<TCoordinate4>     _tex;
        std::vector<TriangularFace>   _face;

    public:
        // special and default constructor
        TriangulatedMesh3(
                GLuint vertex_count = 0, GLuint face_count = 0,
                GLenum usage_flag = GL_STATIC_DRAW);

        // copy constructor
        TriangulatedMesh3(const TriangulatedMesh3& mesh);

        // assignment operator
        TriangulatedMesh3& operator =(const TriangulatedMesh3& rhs);

        // deletes all vertex buffer objects
        GLvoid DeleteVertexBufferObjects();

        // renders the geometry
        GLboolean Render(GLenum render_mode = GL_TRIANGLES) const;

        // updates all vertex buffer objects
        GLboolean UpdateVertexBufferObjects(GLenum usage_flag = GL_STATIC_DRAW);

        // loads the geometry (i.e. the array of vertices and faces) stored in an OFF file
        // at the same time calculates the unit normal vectors associated with vertices
        GLboolean LoadFromOFF(
                const std::string& file_name, GLboolean translate_and_scale_to_unit_cube = GL_FALSE);
```

Triangulated meshes – header file, part III

TriangulatedMeshes3.h

```
        // homework: saves the geometry into an OFF file
        GLboolean SaveToOFF(const std::string& file_name) const;

        // mapping vertex buffer objects
        GLfloat* MapVertexBuffer(GLenum access_flag = GL_READ_ONLY) const;
        GLfloat* MapNormalBuffer(GLenum access_flag = GL_READ_ONLY) const;  // homework
        GLfloat* MapTextureBuffer(GLenum access_flag = GL_READ_ONLY) const; // homework

        // unmapping vertex buffer objects
        GLvoid UnmapVertexBuffer() const;
        GLvoid UnmapNormalBuffer() const;   // homework
        GLvoid UnmapTextureBuffer() const;  // homework

        // properties
        size_t VertexCount() const; // homework
        size_t FaceCount() const;   // homework

        // destructor
        virtual ~TriangulatedMesh3();
    };
}
```

Triangulated meshes – source file, part I

TriangulatedMeshes3.cpp

```cpp
#include <cstring>
#include <fstream>
#include <limits>
#include "TriangulatedMeshes3.h"

using namespace cagd;
using namespace std;

TriangulatedMesh3::TriangulatedMesh3(GLuint vertex_count, GLuint face_count, GLenum usage_flag):
        _usage_flag(usage_flag),
        _vbo_vertices(0), _vbo_normals(0), _vbo_tex_coordinates(0), _vbo_indices(0),
        _vertex(vertex_count), _normal(vertex_count), _tex(vertex_count),
        _face(face_count)
{
}

TriangulatedMesh3::TriangulatedMesh3(const TriangulatedMesh3 &mesh):
        _usage_flag(mesh._usage_flag),
        _vbo_vertices(0), _vbo_normals(0), _vbo_tex_coordinates(0), _vbo_indices(0),
        _leftmost_vertex(mesh._leftmost_vertex), _right_most_vertex(mesh._rightmost_vertex),
        _vertex(mesh._vertex),
        _normal(mesh._normal),
        _tex(mesh._tex),
        _face(mesh._face)
{
    if (mesh._vbo_vertices && mesh._vbo_normals && mesh._vbo_tex_coordinates && mesh._vbo_indices)
        UpdateVertexBufferObjects(mesh._usage_flag);
}

TriangulatedMesh3& TriangulatedMesh3::operator =(const TriangulatedMesh3& rhs)
{
    if (this != &rhs)
    {
        DeleteVertexBufferObjects();
```

Triangulated meshes – source file, part II

TriangulatedMeshes3.cpp

```cpp
        _usage_flag        = rhs._usage_flag;
        _leftmost_vertex   = rhs._leftmost_vertex;
        _rightmost_vertex = rhs._rightmost_vertex;
        _vertex            = rhs._vertex;
        _normal            = rhs._normal;
        _tex               = rhs._tex;
        _face              = rhs._face;

        if (rhs._vbo_vertices && rhs._vbo_normals && rhs._vbo_tex_coordinates && rhs._vbo_indices)
            UpdateVertexBufferObjects(_usage_flag);
    }

    return *this;
}

GLvoid TriangulatedMesh3::DeleteVertexBufferObjects()
{
    if (_vbo_vertices)
    {
        glDeleteBuffers(1, &_vbo_vertices);
        _vbo_vertices = 0;
    }

    // homework: delete vertex buffer objects of unit normal vectors, texture coordinates, and indices
}

GLboolean TriangulatedMesh3::Render(GLenum render_mode) const
{
    if (!_vbo_vertices || !_vbo_normals || !_vbo_tex_coordinates || !_vbo_indices)
        return GL_FALSE;

    if (render_mode != GL_TRIANGLES && render_mode != GL_POINTS)
        return GL_FALSE;

    // enable client states of vertex, normal and texture coordinate arrays
```

Triangulated meshes – source file, part III

TriangulatedMeshes3.cpp

```
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

        // activate the VBO of texture coordinates
        glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
        // specify the location and data format of texture coordinates
        glTexCoordPointer(4, GL_FLOAT, 0, nullptr);

        // activate the VBO of normal vectors
        glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
        // specify the location and data format of normal vectors
        glNormalPointer(GL_FLOAT, 0, nullptr);

        // activate the VBO of vertices
        glBindBuffer(GL_ARRAY_BUFFER, _vbo_vertices);
        // specify the location and data format of vertices
        glVertexPointer(3, GL_FLOAT, 0, nullptr);

        // activate the element array buffer for indexed vertices of triangular faces
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _vbo_indices);

        // render primitives
        glDrawElements(render_mode, static_cast<GLsizei>(3 * _face.size()), GL_UNSIGNED_INT, nullptr);

    // disable individual client-side capabilities
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    // unbind any buffer object previously bound and restore client memory usage
    // for these buffer object targets
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

Triangulated meshes – source file, part IV

TriangulatedMeshes3.cpp

```
    return GL_TRUE;
}

GLboolean TriangulatedMesh3::UpdateVertexBufferObjects(GLenum usage_flag)
{
    if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  && usage_flag != GL_STREAM_COPY
     && usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  && usage_flag != GL_STATIC_COPY
     && usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ && usage_flag != GL_DYNAMIC_COPY)
        return GL_FALSE;

    // updating usage flag
    _usage_flag = usage_flag;

    // deleting old vertex buffer objects
    DeleteVertexBufferObjects();

    // creating vertex buffer objects of mesh vertices, unit normal vectors, texture coordinates,
    // and element indices
    glGenBuffers(1, &_vbo_vertices);

    if (!_vbo_vertices)
        return GL_FALSE;

    glGenBuffers(1, &_vbo_normals);

    if (!_vbo_normals)
    {
        glDeleteBuffers(1, &_vbo_vertices);
        _vbo_vertices = 0;
        return GL_FALSE;
    }

    glGenBuffers(1, &_vbo_tex_coordinates);
    if (!_vbo_tex_coordinates)
    {
```

Triangulated meshes – source file, part V

TriangulatedMeshes3.cpp

```
        glDeleteBuffers(1, &_vbo_vertices);
        _vbo_vertices = 0;

        glDeleteBuffers(1, &_vbo_normals);
        _vbo_normals = 0;

        return GL_FALSE;
    }

    glGenBuffers(1, &_vbo_indices);
    if (!_vbo_indices)
    {
        glDeleteBuffers(1, &_vbo_vertices);
        _vbo_vertices = 0;

        glDeleteBuffers(1, &_vbo_normals);
        _vbo_normals = 0;

        glDeleteBuffers(1, &_vbo_tex_coordinates);
        _vbo_tex_coordinates = 0;

        return GL_FALSE;
    }

    // For efficiency reasons we convert all GLdouble coordinates
    // to GLfloat coordinates: we will use auxiliar pointers for
    // buffer data loading, by means of the functions glMapBuffer/glUnmapBuffer.

    // Notice that multiple buffers can be mapped simultaneously.

    size_t vertex_byte_size = 3 * _vertex.size() * sizeof(GLfloat);

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_vertices);
    glBufferData(GL_ARRAY_BUFFER, vertex_byte_size, nullptr, _usage_flag);
```

Triangulated meshes – source file, part VI

TriangulatedMeshes3.cpp

```
GLfloat *vertex_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
glBufferData(GL_ARRAY_BUFFER, vertex_byte_size, nullptr, _usage_flag);

GLfloat *normal_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

for (vector<DCoordinate3>::const_iterator
     vit = _vertex.begin(),
     nit = _normal.begin(); vit != _vertex.end(); ++vit, ++nit)
{
    for (GLint component = 0; component < 3; ++component)
    {
        *vertex_coordinate = (GLfloat)(*vit)[component];
        ++vertex_coordinate;

        *normal_coordinate = (GLfloat)(*nit)[component];
        ++normal_coordinate;
    }
}

size_t tex_byte_size = 4 * _tex.size() * sizeof(GLfloat);

glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
glBufferData(GL_ARRAY_BUFFER, tex_byte_size, nullptr, _usage_flag);
GLfloat *tex_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

memcpy(tex_coordinate, &_tex[0][0], tex_byte_size);

size_t index_byte_size = 3 * _face.size() * sizeof(GLuint);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _vbo_indices);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, index_byte_size, nullptr, _usage_flag);
GLuint *element = (GLuint*)glMapBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY);
```

Triangulated meshes – source file, part VII

TriangulatedMeshes3.cpp

```
    for (vector<TriangularFace>::const_iterator fit = _face.begin(); fit != _face.end(); ++fit)
    {
        for (GLint node = 0; node < 3; ++node)
        {
            *element = (*fit)[node];
            ++element;
        }
    }

    // unmap all VBOs
    glBindBuffer(GL_ARRAY_BUFFER, _vbo_vertices);
    if (!glUnmapBuffer(GL_ARRAY_BUFFER))
        return GL_FALSE;

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
    if (!glUnmapBuffer(GL_ARRAY_BUFFER))
        return GL_FALSE;

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
    if (!glUnmapBuffer(GL_ARRAY_BUFFER))
        return GL_FALSE;

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _vbo_indices);
    if (!glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER))
        return GL_FALSE;

    // unbind any buffer object previously bound and restore client memory usage
    // for these buffer object targets
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

    return GL_TRUE;
}
```

Triangulated meshes – source file, part VIII

TriangulatedMeshes3.cpp

```
GLboolean TriangulatedMesh3::LoadFromOFF(
        const string &file_name, GLboolean translate_and_scale_to_unit_cube)
{
    fstream f(file_name.c_str(), ios_base::in);

    if (!f || !f.good())
        return GL_FALSE;

    // loading the header
    string header;

    f >> header;

    if (header != "OFF")
        return GL_FALSE;

    // loading number of vertices, faces, and edges
    GLuint vertex_count, face_count, edge_count;

    f >> vertex_count >> face_count >> edge_count;

    // allocating memory for vertices, unit normal vectors, texture coordinates, and faces
    _vertex.resize(vertex_count);
    _normal.resize(vertex_count);
    _tex.resize(vertex_count);
    _face.resize(face_count);

    // initializing the leftmost and rightmost corners of the bounding box
    _leftmost_vertex.x() = _leftmost_vertex.y() = _leftmost_vertex.z()
                                                = numeric_limits<GLdouble>::max();
    _rightmost_vertex.x() = _rightmost_vertex.y() = _rightmost_vertex.z()
                                                  = -numeric_limits<GLdouble>::max();
```

Triangulated meshes – source file, part IX

TriangulatedMeshes3.cpp

```cpp
    // loading vertices and correcting the leftmost and rightmost corners of the bounding box
    for (vector<DCoordinate3>::iterator vit = _vertex.begin(); vit != _vertex.end(); ++vit)
    {
        f >> *vit;

        if (vit->x() < _leftmost_vertex.x())
            _leftmost_vertex.x() = vit->x();
        if (vit->y() < _leftmost_vertex.y())
            _leftmost_vertex.y() = vit->y();
        if (vit->z() < _leftmost_vertex.z())
            _leftmost_vertex.z() = vit->z();

        if (vit->x() > _rightmost_vertex.x())
            _rightmost_vertex.x() = vit->x();
        if (vit->y() > _rightmost_vertex.y())
            _rightmost_vertex.y() = vit->y();
        if (vit->z() > _rightmost_vertex.z())
            _rightmost_vertex.z() = vit->z();
    }

    // if we do not want to preserve the original positions and coordinates of vertices
    if (translate_and_scale_to_unit_cube)
    {
        GLdouble scale =1.0 / max(_rightmost_vertex.x() - _leftmost_vertex.x(),
                                  max(_rightmost_vertex.y() - _leftmost_vertex.y(),
                                      _rightmost_vertex.z() - _leftmost_vertex.z()));
        DCoordinate3 middle(_leftmost_vertex);
        middle += _rightmost_vertex;
        middle *= 0.5;
        for (vector<DCoordinate3>::iterator vit = _vertex.begin(); vit != _vertex.end(); ++vit)
        {
            *vit -= middle;
            *vit *= scale;
        }
    }
```

Triangulated meshes – source file, part X

TriangulatedMeshes3.cpp

```cpp
    // loading faces
    for (vector<TriangularFace>::iterator fit = _face.begin(); fit != _face.end(); ++fit)
        f >> *fit;

    // calculating average unit normal vectors associated with vertices
    for (vector<TriangularFace>::const_iterator fit = _face.begin(); fit != _face.end(); ++fit)
    {
        DCoordinate3 n = _vertex[(*fit)[1]];
        n -= _vertex[(*fit)[0]];

        DCoordinate3 p = _vertex[(*fit)[2]];
        p -= _vertex[(*fit)[0]];

        n ^= p;

        for (GLint node = 0; node < 3; ++node)
            _normal[(*fit)[node]] += n;
    }

    for (vector<DCoordinate3>::iterator nit = _normal.begin(); nit != _normal.end(); ++nit)
        nit->normalize();

    f.close();

    return GL_TRUE;
}

GLfloat* TriangulatedMesh3::MapVertexBuffer(GLenum access_flag) const
{
    if (access_flag != GL_READ_ONLY && access_flag != GL_WRITE_ONLY && access_flag != GL_READ_WRITE)
        return (GLfloat*)0;

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_vertices);
    GLfloat* result = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, access_flag);
```

Triangulated meshes – source file, part XI

TriangulatedMeshes3.cpp

```
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    return result;
}

GLvoid TriangulatedMesh3::UnmapVertexBuffer() const
{
    glBindBuffer(GL_ARRAY_BUFFER, _vbo_vertices);
    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

TriangulatedMesh3::~TriangulatedMesh3()
{
    DeleteVertexBufferObjects();
}
```

Testing dynamic vertex buffer objects

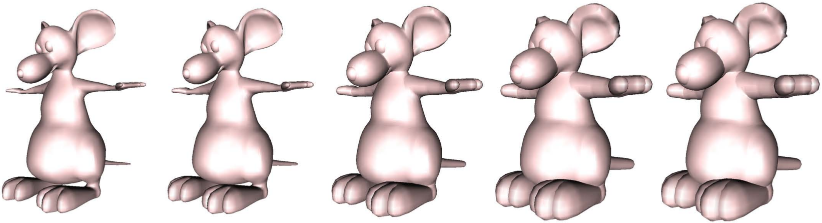


Fig. 1: How can we achieve such an effect?

Testing dynamic vertex buffer objects

GLWidget.h

```
#pragma once

#include <GL/glew.h>
#include <QOpenGLWidget>
#include <QTimer>
#include "../Core/TriangulatedMeshes3.h"

namespace cagd
{
    class GLWidget: public QOpenGLWidget
    {
        Q_OBJECT

    private:
        QTimer *_timer;
        GLfloat _angle;

        ...

        TriangulatedMesh3 _mouse;

        ...

    private slots:
        void _animate();

    public:

        ...
    };
}
```

Testing dynamic vertex buffer objects, part I

GLWidget.cpp

```cpp
#include "GLWidget.h"
#include "Materials.h"

using namespace cagd;
using namespace std;

GLWidget::GLWidget(QWidget *parent): QOpenGLWidget(parent)
{
    ...

    _timer = new QTimer(this);
    _timer->setInterval(0);

    connect(_timer, SIGNAL(timeout()), this, SLOT(_animate()));

    ...
}

void GLWidget::initializeGL()
{
    ...

    try
    {
        // initializing the OpenGL Extension Wrangler library
        GLenum error = glewInit();

        if (error != GLEW_OK)
        {
            throw Exception("Could␣not␣initialize␣the␣OpenGL␣Extension␣Wrangler␣Library!");
        }

        if (!glewIsSupported("GL_VERSION_2_0"))
        {
            throw Exception("Your␣graphics␣card␣is␣not␣compatible␣with␣OpenGL␣2.0+!␣"
```

Testing dynamic vertex buffer objects, part II

GLWidget.cpp

```
            "Try to update your driver or buy a new graphics adapter!");
        }

        if (!_mouse.LoadFromOFF("Models/Characters/mouse.off", GL_TRUE))
        {
            throw Exception("Could not load the model file!");
        }

        if (!_mouse.UpdateVertexBufferObjects(GL_DYNAMIC_DRAW))
        {
            throw Exception("Could not update the vertex buffer objects of the triangulated mesh!");
        }
        _angle = 0.0;
        _timer->start();
    }
    catch (Exception &e)
    {
        cout << e << endl;
    }

    ...
}

void GLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ...
    glPushMatrix();
        ...
        MatFBPearl.Apply();
        _mouse.Render();
        ...
    glPopMatrix();
}
```

Testing dynamic vertex buffer objects, part III

GLWidget.cpp

```
void GLWidget::_animate()
{
    GLfloat *vertex = _mouse.MapVertexBuffer(GL_READ_WRITE);
    GLfloat *normal = _mouse.MapNormalBuffer(GL_READ_ONLY);

    _angle += DEG_TO_RADIAN;
    if (_angle >= TWO_PI) _angle -= TWO_PI;

    GLfloat scale = sin(_angle) / 3000.0;
    for (GLuint i = 0; i < _mouse.VertexCount(); ++i)
    {
        for (GLuint coordinate = 0; coordinate < 3; ++coordinate, ++vertex, ++normal)
            *vertex += scale * (*normal);
    }

    _mouse.UnmapVertexBuffer();
    _mouse.UnmapNormalBuffer();

    update();
}
```

Colors and intensities – header file, part I

Colors4.h, ∄ Colors4.cpp

```
#pragma once

#include <GL/glew.h>

namespace cagd
{
    class Color4
    {
    protected:
        GLfloat _data[4]; // (r, g, b, a)

    public:
        Color4()
        {
            _data[0] = _data[1] = _data[2] = 0.0;
            _data[3] = 1.0;
        }

        Color4(GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f)
        {
            _data[0] = r;
            _data[1] = g;
            _data[2] = b;
            _data[3] = a;
        }

        // homework: get components by value
        GLfloat operator [](GLuint rhs) const;
        GLfloat r() const;
        GLfloat g() const;
        GLfloat b() const;
        GLfloat a() const;

        // homework: get components by reference
        GLfloat& operator [](GLuint rhs);
```

Colors and intensities – header file, part II

Colors4.h, ∄ Colors4.cpp

```
        GLfloat& r();
        GLfloat& g();
        GLfloat& b();
        GLfloat& a();
    };
}
```

Lights

Classes DirectionalLight, PointLight, and Spotlight

Description

- OpenGL can handle directional, point and spotlights. A scene may have at most 8 different light sources.
- Light sources have several properties, such as ambient, diffuse and specular intensities, position, direction, and possible constant, linear and quadratic attenuation.
- Light sources can be handled by means of commands:

```
GLvoid glLight{i|f}(GLenum light_index, GLenum parameter_name, {int|float} value);
GLvoid glLight{i|f}v(GLenum light_index, GLenum parameter_name, {int|float} *vector);
```

Description – continued

```
/*
   light_index can be GL_LIGHT0, GL_LIGHT1, ..., or GL_LIGHT7.
   The characteristic of the light being set is defined by parameter_name,
   which specifies a named parameter from the table below.

   Parameter name              |    Default value          |   Meaning

  GL_AMBIENT                   |  (0.0, 0.0, 0.0, 1.0)     |   ambient intensity of the light

  GL_DIFFUSE                   |  (1.0, 1.0, 1.0, 1.0)     |   diffuse intensity of the light
                               |           or              |   (default for light 0 is white;
                               |  (0.0, 0.0, 0.0, 1.0)     |   for other lights, black

  GL_SPECULAR                  |  (1.0, 1.0, 1.0, 1.0)     |   specular intensity of the light
                               |           or              |   (default for light 0 is white;
                               |  (0.0, 0.0, 0.0, 1.0)     |   for other lights, black

  GL_POSITION                  |  (0.0, 0.0, 1.0, 0.0)     |   (x, y, z, w) position of light

  GL_SPOT_DIRECTION            |  (0.0, 0.0, -1.0)         |   (x, y, z) direction of spotlight

  GL_SPOT_EXPONENT             |  0.0                      |   spotlight exponent

  GL_SPOT_CUTOFF               |  180.0                    |   spotlight cutoff angle

  GL_CONSTANT_ATTENUATION      |  1.0                      |   constant attenuation factor

  GL_LINEAR_ATTENUATION        |  0.0                      |   linear attenuation factor

  GL_QUADRATIC_ATTENUATION     |  0.0                      |   quadratic attenuation factor

   Variables value and vector indicate values to which the parameter_name characteristic is set:
   it is the value itself if the non-vector version is used or it is a pointer to a group of values
   if the vector version is used.
*/
```

Lights – header file, part I

Lights.h

```
#pragma once

#include "Colors4.h"
#include "HCoordinates3.h"
#include <GL/glew.h>

namespace cagd
{
    class DirectionalLight
    {
    protected:
        GLenum       _light_index;
        HCoordinate3 _position;
        Color4       _ambient_intensity, _diffuse_intensity, _specular_intensity;

    public:
        DirectionalLight(
            GLenum               light_index,
            const HCoordinate3&  position,
            const Color4&        ambient_intensity,
            const Color4&        diffuse_intensity,
            const Color4&        specular_intensity);

        void Enable();
        void Disable();
    };

    class PointLight: public DirectionalLight
    {
    protected:
        GLfloat _constant_attenuation,
                _linear_attenuation,
                _quadratic_attenuation;
```

Lights – header file, part II

Lights.h

```
    public:
        PointLight(
            GLenum              light_index,
            const HCoordinate3& position,
            const Color4&       ambient_intensity,
            const Color4&       diffuse_intensity,
            const Color4&       specular_intensity,
            GLfloat             constant_attenuation,
            GLfloat             linear_attenuation,
            GLfloat             quadratic_attenuation);
    };

    class Spotlight: public PointLight
    {
    private:
        HCoordinate3 _spot_direction;
        GLfloat      _spot_cutoff, _spot_exponent;

    public:
        Spotlight(
            GLenum              light_index,
            const HCoordinate3& position,
            const Color4&       ambient_intensity,
            const Color4&       diffuse_intensity,
            const Color4&       specular_intensity,
            GLfloat             constant_attenuation,
            GLfloat             linear_attenuation,
            GLfloat             quadratic_attenuation,
            const HCoordinate3& spot_direction,
            GLfloat             spot_cutoff,
            GLfloat             spot_exponent);
    };
}
```

Lights – source file, part I

Lights.cpp

```
#include "Exceptions.h"
#include "Lights.h"

using namespace cagd;

DirectionalLight::DirectionalLight(
    GLenum               light_index,
    const HCoordinate3&  position,
    const Color4&        ambient_intensity,
    const Color4&        diffuse_intensity,
    const Color4&        specular_intensity):

    _light_index(light_index),
    _position(position),
    _ambient_intensity(ambient_intensity),
    _diffuse_intensity(diffuse_intensity),
    _specular_intensity(specular_intensity)
{
    glLightfv(light_index, GL_POSITION, &_position.x());
    glLightfv(light_index, GL_AMBIENT,  &_ambient_intensity.r());
    glLightfv(light_index, GL_DIFFUSE,  &_diffuse_intensity.r());
    glLightfv(light_index, GL_SPECULAR, &_specular_intensity.r());
}

void DirectionalLight::Enable()
{
    glEnable(_light_index);
}

void DirectionalLight::Disable()
{
    glDisable(_light_index);
}
```

Lights – source file, part II

Lights.cpp

```cpp
PointLight::PointLight(
    GLenum              light_index,
    const HCoordinate3& position,
    const Color4&       ambient_intensity,
    const Color4&       diffuse_intensity,
    const Color4&       specular_intensity,
    GLfloat             constant_attenuation,
    GLfloat             linear_attenuation,
    GLfloat             quadratic_attenuation):

    DirectionalLight(
            light_index,
            position,
            ambient_intensity, diffuse_intensity, specular_intensity),
    _constant_attenuation(constant_attenuation),
    _linear_attenuation(linear_attenuation),
    _quadratic_attenuation(quadratic_attenuation)
{
    if (position.w() == 0.0)
        throw Exception("PointLight::PointLight - Wrong position.");

    glLightf(_light_index, GL_SPOT_CUTOFF, 180.0);
    glLightf(_light_index, GL_CONSTANT_ATTENUATION,  _constant_attenuation);
    glLightf(_light_index, GL_LINEAR_ATTENUATION,    _linear_attenuation);
    glLightf(_light_index, GL_QUADRATIC_ATTENUATION, _quadratic_attenuation);
}
```

Lights – source file, part III

Lights.cpp

```
Spotlight::Spotlight(
    GLenum                  light_index,
    const HCoordinate3& position,
    const Color4&       ambient_intensity,
    const Color4&       diffuse_intensity,
    const Color4&       specular_intensity,
    GLfloat             constant_attenuation,
    GLfloat             linear_attenuation,
    GLfloat             quadratic_attenuation,
    const HCoordinate3& spot_direction,
    GLfloat             spot_cutoff,
    GLfloat             spot_exponent):
    PointLight(
            light_index, position,
            ambient_intensity, diffuse_intensity, specular_intensity,
            constant_attenuation, linear_attenuation, quadratic_attenuation),
    _spot_direction(spot_direction),
    _spot_cutoff(spot_cutoff),
    _spot_exponent(spot_exponent)
{
    if (position.w() == 0.0)
        throw Exception("Spotlight::Spotlight - Wrong position.");

    if (_spot_cutoff > 90.0)
        throw Exception("Spotlight::Spotlight - Wrong spot cutoff.");

    glLightfv(_light_index, GL_SPOT_DIRECTION, &_spot_direction.x());
    glLightf (_light_index, GL_SPOT_CUTOFF,     _spot_cutoff);
    glLightf (_light_index, GL_SPOT_EXPONENT,   _spot_exponent);
}
```

Lights – usage, part I

Example

```
#pragma once

#include <GL/glew.h>
#include <QOpenGLWidget>
#include "../Core/Lights.h"

namespace cagd
{
    class GLWidget: public QOpenGLWidget
    {
        Q_OBJECT

    private:

        ...

        DirectionalLight *_dl = nullptr; // pointer to a directional light object

        ...

    public:

        ...

    };
}
```

Lights – usage, part II

Example

```
#include "../Core/Colors4.h"
#include "../Core/HCoordinates3.h"


using namespace cagd;

...

void GLWidget::initializeGL()
{
    ...

    try
    {
        // initializing the OpenGL Extension Wrangler library
        GLenum error = glewInit();

        if (error != GLEW_OK)
        {
            throw Exception("Could not initialize the OpenGL Extension Wrangler Library!");
        }

        if (!glewIsSupported("GL_VERSION_2_0"))
        {
            throw Exception("Your graphics card is not compatible with OpenGL 2.0+! "
                            "Try to update your driver or buy a new graphics adapter!");
        }

        ...

        // creating a white directional light source
        HCoordinate3 direction(0.0, 0.0, 1.0, 0.0);
        Color4       ambient  (0.4, 0.4, 0.4, 1.0);
        Color4       diffuse  (0.8, 0.8, 0.8, 1.0);
        Color4       specular (1.0, 1.0, 1.0, 1.0);
```

Lights – usage, part III

Example

```
        _dl = new (nothrow) DirectionalLight(GL_LIGHT0, direction, ambient, diffuse, specular);

        if (!_dl)
        {
            throw Exception("Could␣not␣create␣the␣directional␣light␣object!");
        }

        glEnable(GL_LIGHTING);
        glEnable(GL_NORMALIZE);
    }
    catch (Exception &e)
    {
        cout << e << endl;
    }
}

void GLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ...
    glPushMatrix();
    ...
        if (_dl)
        {
            _dl->Enable();

            // render geometry with unit normal vectors
            ...

            _dl->Disable();
        }
    ...
    glPopMatrix();
}
```

Lights – usage, part IV

Example

```
void GLWidget::~GLWidget()
{
    // free the allocated memory of the light source
    if (_dl)
    {
        delete _dl; _dl = nullptr;
    }
}
```

Materials

Class Material

Descpription

- The commands

```
/*
    face can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate to which faces of the object
    the material should be applied.
    The particular material characteristic (GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION,
    GL_SHININESS) being set is identified by parameter_name, and the desired value(s) for that
    charecteristic is identified either by value or vector.
*/

void glMaterial{i|f}(GLenum face, GLenum parameter_name, {int|float} value);
void glMaterial{i|f}v(GLenum face, GLenum parameter_name, {int|float} *vector);
```

specify the current material property for use in lighting calculations.

Materials – header file, part I

Materials.h

```
#pragma once

#include "Colors4.h"
#include <GL/glew.h>

namespace cagd
{
    class Material
    {
    protected:
        Color4  _front_ambient, _front_diffuse, _front_specular, _front_emissive;
        GLfloat _front_shininess;

        Color4  _back_ambient, _back_diffuse, _back_specular, _back_emissive;
        GLfloat _back_shininess;

    public:
        Material(
            const Color4& front_ambient   = Color4(),
            const Color4& front_diffuse   = Color4(),
            const Color4& front_specular  = Color4(),
            const Color4& front_emissive  = Color4(),
            GLfloat front_shininess       = 128.0f,
            const Color4& backAmbient     = Color4(),
            const Color4& back_diffuse    = Color4(),
            const Color4& back_specular   = Color4(),
            const Color4& back_emissive   = Color4(),
            GLfloat back_shininess        = 128.0f);

        GLvoid SetAmbientColor(GLenum face, const Color4& c);
        GLvoid SetAmbientColor(GLenum face, GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

        // homework
        GLvoid SetDiffuseColor(GLenum face, const Color4& c);
        GLvoid SetDiffuseColor(GLenum face, GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);
```

Materials – header file, part II

Materials.h

```
        // homework
        GLvoid SetSpecularColor(GLenum face, const Color4& c);
        GLvoid SetSpecularColor(GLenum face, GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

        // homework
        GLvoid SetEmissiveColor(GLenum face, const Color4& c);
        GLvoid SetEmissiveColor(GLenum face, GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

        // homework
        GLvoid SetShininess(GLenum face, GLfloat shininess);

        // homework
        GLvoid SetTransparency(GLfloat alpha);

        GLvoid Apply();

        // homework
        GLboolean IsTransparent() const;
    };

    extern
    Material     MatFBBrass,
                 MatFBGold,
                 MatFBSilver,
                 MatFBEmerald,
                 MatFBPearl,
                 MatFBRuby,
                 MatFBTurquoise;
}
```

Materials – source file, part I

Materials.cpp

```cpp
#include "Materials.h"

using namespace cagd;

Material::Material(
    const Color4& front_ambient,
    const Color4& front_diffuse,
    const Color4& front_specular,
    const Color4& front_emissive,
    GLfloat front_shininess,

    const Color4& back_ambient,
    const Color4& back_diffuse,
    const Color4& back_specular,
    const Color4& back_emissive,
    GLfloat back_shininess):

    _front_ambient      (front_ambient),
    _front_diffuse      (front_diffuse),
    _front_specular     (front_specular),
    _front_emissive     (front_emissive),
    _front_shininess    (front_shininess),

    _back_ambient       (back_ambient),
    _back_diffuse       (back_diffuse),
    _back_specular      (back_specular),
    _back_emissive      (back_emissive),
    _back_shininess     (back_shininess)
{
}
```

Materials – source file, part II

Materials.cpp

```
GLvoid Material::SetAmbientColor(GLenum face, GLfloat r, GLfloat g, GLfloat b, GLfloat a)
{
    switch (face)
    {
    case GL_FRONT:
        _front_ambient.r() = r;
        _front_ambient.g() = g;
        _front_ambient.b() = b;
        _front_ambient.a() = a;
    break;

    case GL_BACK:
        _back_ambient.r() = r;
        _back_ambient.g() = g;
        _back_ambient.b() = b;
        _back_ambient.a() = a;
    break;

    case GL_FRONT_AND_BACK:
        _front_ambient.r() = r;
        _front_ambient.g() = g;
        _front_ambient.b() = b;
        _front_ambient.a() = a;

        _back_ambient.r()  = r;
        _back_ambient.g()  = g;
        _back_ambient.b()  = b;
        _back_ambient.a()  = a;
    break;
    }
}
```

Materials – source file, part III

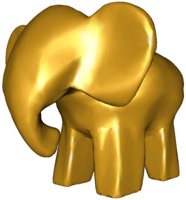
Materials.cpp

```
GLvoid Material::Apply()
{
    glMaterialfv(GL_FRONT, GL_AMBIENT,   &_front_ambient.r());
    glMaterialfv(GL_FRONT, GL_DIFFUSE,   &_front_diffuse.r());
    glMaterialfv(GL_FRONT, GL_SPECULAR,  &_front_specular.r());
    glMaterialfv(GL_FRONT, GL_EMISSION,  &_front_emissive.r());
    glMaterialf (GL_FRONT, GL_SHININESS, _front_shininess);

    glMaterialfv(GL_BACK, GL_AMBIENT,    &_back_ambient.r());
    glMaterialfv(GL_BACK, GL_DIFFUSE,    &_back_diffuse.r());
    glMaterialfv(GL_BACK, GL_SPECULAR,   &_back_specular.r());
    glMaterialfv(GL_BACK, GL_EMISSION,   &_back_emissive.r());
    glMaterialf (GL_BACK, GL_SHININESS,  _back_shininess);
}
```

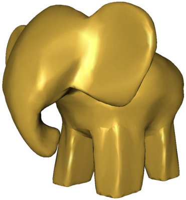
Materials – source file, part IV

Materials.cpp

```
// brass
Material cagd::MatFBBrass = Material(
                        Color4(0.329412f, 0.223529f, 0.027451f, 0.4f),
                        Color4(0.780392f, 0.568627f, 0.113725f, 0.6f),
                        Color4(0.992157f, 0.941176f, 0.807843f, 0.8f),
                        Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                        27.8974f,
                        Color4(0.329412f, 0.223529f, 0.027451f, 0.4f),
                        Color4(0.780392f, 0.568627f, 0.113725f, 0.6f),
                        Color4(0.992157f, 0.941176f, 0.807843f, 0.8f),
                        Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                        27.8974f);
```



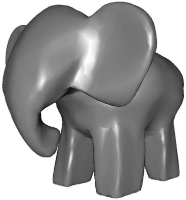
```
// gold
Material cagd::MatFBGold = Material(
                        Color4(0.247250f, 0.199500f, 0.074500f, 0.4f),
                        Color4(0.751640f, 0.606480f, 0.226480f, 0.6f),
                        Color4(0.628281f, 0.555802f, 0.366065f, 0.8f),
                        Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                        51.2f,
                        Color4(0.247250f, 0.199500f, 0.074500f, 0.4f),
                        Color4(0.751640f, 0.606480f, 0.226480f, 0.6f),
                        Color4(0.628281f, 0.555802f, 0.366065f, 0.8f),
                        Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                        51.2f);
```



Materials – source file, part V

Materials.cpp

```
// silver
Material cagd::MatFBSilver = Material(
                                Color4(0.192250f, 0.192250f, 0.192250f, 0.4f),
                                Color4(0.507540f, 0.507540f, 0.507540f, 0.6f),
                                Color4(0.508273f, 0.508273f, 0.508273f, 0.8f),
                                Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                51.2f,
                                Color4(0.192250f, 0.192250f, 0.192250f, 0.4f),
                                Color4(0.507540f, 0.507540f, 0.507540f, 0.6f),
                                Color4(0.508273f, 0.508273f, 0.508273f, 0.8f),
                                Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                51.2f);
```

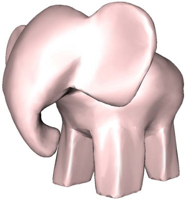


```
// emerald
Material cagd::MatFBEmerald = Material(
                                Color4(0.021500f, 0.174500f, 0.021500f, 0.4f),
                                Color4(0.075680f, 0.614240f, 0.075680f, 0.6f),
                                Color4(0.633000f, 0.727811f, 0.633000f, 0.8f),
                                Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                76.8f,
                                Color4(0.021500f, 0.174500f, 0.021500f, 0.4f),
                                Color4(0.075680f, 0.614240f, 0.075680f, 0.6f),
                                Color4(0.633000f, 0.727811f, 0.633000f, 0.8f),
                                Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                76.8f);
```

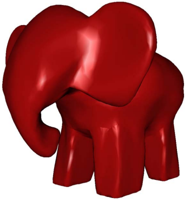
Materials – source file, part VI

Materials.cpp

```
// pearl
Material cagd::MatFBPearl = Material(
                                     Color4(0.250000f, 0.207250f, 0.207250f, 0.4f),
                                     Color4(1.000000f, 0.829000f, 0.829000f, 0.6f),
                                     Color4(0.296648f, 0.296648f, 0.296648f, 0.8f),
                                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                     11.264f,
                                     Color4(0.250000f, 0.207250f, 0.207250f, 0.4f),
                                     Color4(1.000000f, 0.829000f, 0.829000f, 0.6f),
                                     Color4(0.296648f, 0.296648f, 0.296648f, 0.8f),
                                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                     11.264f);
```



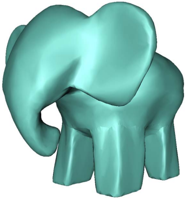
```
// ruby
Material cagd::MatFBRuby = Material(
                                     Color4(0.174500f, 0.011750f, 0.011750f, 0.4f),
                                     Color4(0.614240f, 0.041360f, 0.041360f, 0.6f),
                                     Color4(0.727811f, 0.626959f, 0.626959f, 0.8f),
                                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                     76.8f,
                                     Color4(0.174500f, 0.011750f, 0.011750f, 0.4f),
                                     Color4(0.614240f, 0.041360f, 0.041360f, 0.6f),
                                     Color4(0.727811f, 0.626959f, 0.626959f, 0.8f),
                                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                                     76.8f);
```



Materials – source file, part VII

Materials.cpp

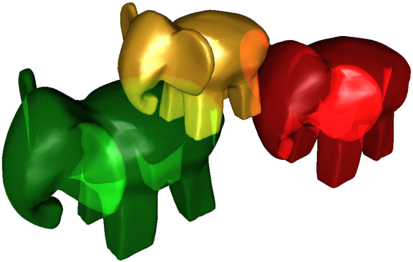
```
// turquoise
Material cagd::MatFBTurquoise = Material(
                        Color4(0.100000f, 0.187250f, 0.174500f, 0.4f),
                        Color4(0.396000f, 0.741510f, 0.691020f, 0.6f),
                        Color4(0.297254f, 0.308290f, 0.306678f, 0.8f),
                        Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                        12.8f,
                        Color4(0.100000f, 0.187250f, 0.174500f, 0.4f),
                        Color4(0.396000f, 0.741510f, 0.691020f, 0.6f),
                        Color4(0.297254f, 0.308290f, 0.306678f, 0.8f),
                        Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
                        12.8f);
```



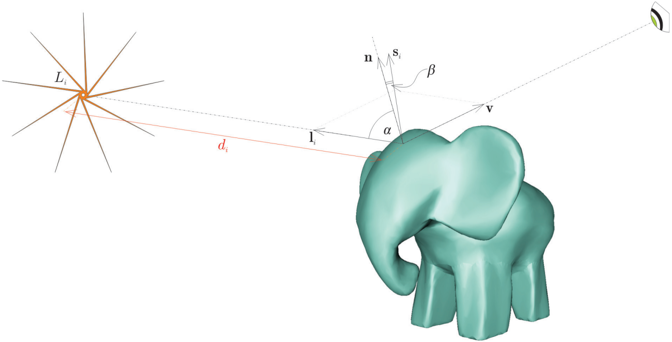
Transparency

Example

```
void GLWidget::paintGL()
{
    ...

    glPushMatrix();
        // transformations
        MatFBBrass.Apply();
        _mesh.Render();
    glPopMatrix();

    glEnable(GL_BLEND);

        glDepthMask(GL_FALSE);

            glBlendFunc(GL_SRC_ALPHA, GL_ONE);

            glPushMatrix();
                // transformations
                MatFBRuby.Apply();
                _mesh.Render();
            glPopMatrix();

            glPushMatrix();
                // transformations
                MatFBEmerald.Apply();
                _mesh.Render();
            glPopMatrix();

        glDepthMask(GL_TRUE);

    glDisable(GL_BLEND);

    ...
}
```



The mathematics of lighting



$$
\begin{aligned}
\text{vertex color} \quad = \quad & \sum_{i=0}^{n-1} \frac{1}{L_{i,\textit{constant attenuation}} + L_{i,\textit{linear attenuation}} d_i + L_{i,\textit{quadratic attenuation}} d_i^2} \\
& \cdot L_{i,\textit{spotlight effect}} \cdot \Big[L_{i,\textit{ambient}} \odot M_{\textit{ambient}} \\
& + \max\left\{\left\langle \mathbf{l}_i, \mathbf{n} \right\rangle, 0\right\} \cdot L_{i,\textit{diffuse}} \odot M_{\textit{diffuse}} \\
& + \max\left\{\left\langle \mathbf{s}_i, \mathbf{n} \right\rangle, 0\right\}^{L_{i,\textit{shininess}}} \cdot L_{i,\textit{specular}} \odot M_{\textit{specular}} \Big], \ \|\mathbf{l}_i\| = \|\mathbf{s}_i\| = \|\mathbf{n}\| = 1, \\
& \mathbf{s}_i = \frac{\mathbf{l}_i + \mathbf{v}}{\|\mathbf{l}_i + \mathbf{v}\|}.
\end{aligned}
$$