Curve and surface modeling

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

Ágoston Róth

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

(agoston.roth@gmail.com)

Lecture 3 – March 14, 2022

Notations

Used colors

Color styles

- keywords, built-in types, enumerations, constants and namespaces of C++
- keywords, built-in types, enumerations, constants and functions of OpenGL
- our types, constants, enumerations and namespaces
- comments

Descartes coordinates

Class DCoordinate3

Description

- Class DCoordinate3 forms the most elementary building block/data structure of all other classes.
- By means of Descartes coordinates we can represent curve points, tangent and acceleration vectors (i.e. higher order derivatives) of curves, surface points, higher order (mixed) partial derivatives of surfaces, normal vectors associated with surface points, control polygons, control nets, etc.
- Using operator overloading we can easily implement mathematical formulas that describe either curves or surfaces.

Descartes coordinates – header file, part I

DCoordinates3.h

```
#pragma once

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

namespace cagd
{
    //--------------------------
    // class DCoordinate3
    //--------------------------
    class DCoordinate3
    {
    private:
        GLdouble _data[3];

    public:

        // default constructor
        DCoordinate3();

        // special constructor
        DCoordinate3(GLdouble x, GLdouble y, GLdouble z = 0.0);

        // get components by value
        const GLdouble operator [](GLuint index) const;
        const GLdouble x() const;
        const GLdouble y() const;
        const GLdouble z() const;


        // get components by reference
        GLdouble& operator [](GLuint index);
        GLdouble& x();
        GLdouble& y();
```

Descartes coordinates – header file, part II

DCoordinates3.h

```
        GLdouble& z();

        // change sign
        const DCoordinate3 operator +() const;
        const DCoordinate3 operator -() const;

        // add
        const DCoordinate3 operator +(const DCoordinate3& rhs) const;

        // add to *this
        DCoordinate3& operator +=(const DCoordinate3& rhs);

        // subtract
        const DCoordinate3 operator -(const DCoordinate3& rhs) const;

        // subtract from *this
        DCoordinate3& operator -=(const DCoordinate3& rhs);

        // cross product
        const DCoordinate3 operator ^(const DCoordinate3& rhs) const;

        // cross product, result is stored by *this
        DCoordinate3& operator ^=(const DCoordinate3& rhs);

        // dot product
        const GLdouble operator *(const DCoordinate3& rhs) const;

        // scale
        const DCoordinate3 operator *(const GLdouble& rhs) const;
        const DCoordinate3 operator /(const GLdouble& rhs) const;

        // scale *this
        DCoordinate3& operator *=(const GLdouble& rhs);
        DCoordinate3& operator /=(const GLdouble& rhs);
```

Descartes coordinates – header file, part III

DCoordinates3.h

```
    // length
    const GLdouble length() const;

    // normalize
    DCoordinate3& normalize();
};

//----------------------------------------------------------------------
// implementation of class DCoordinate3
//----------------------------------------------------------------------

// default constructor
inline DCoordinate3::DCoordinate3()
{
    _data[0] = _data[1] = _data[2] = 0.0;
}

// special constructor
inline DCoordinate3::DCoordinate3(GLdouble x, GLdouble y, GLdouble z)
{
    _data[0] = x;
    _data[1] = y;
    _data[2] = z;
}

// get components by value
inline const GLdouble DCoordinate3::operator [](GLuint index) const
{
    return _data[index];
}

inline const GLdouble DCoordinate3::x() const
{
    return _data[0];
}
```

Descartes coordinates – header file, part IV

DCoordinates3.h

```
    inline const GLdouble DCoordinate3::y() const
    {
        // homework
    }

    inline const GLdouble DCoordinate3::z() const
    {
        // homework
    }

    // get components by reference
    inline GLdouble& DCoordinate3::operator [](GLuint index)
    {
        return _data[index];
    }

    inline GLdouble& DCoordinate3::x()
    {
        return _data[0];
    }

    inline GLdouble& DCoordinate3::y()
    {
        // homework
    }

    inline GLdouble& DCoordinate3::z()
    {
        // homework
    }

    // change sign
    inline const DCoordinate3 DCoordinate3::operator +() const
    {
```

Descartes coordinates – header file, part V

DCoordinates3.h

```cpp
        return DCoordinate3(_data[0], _data[1], _data[2]);
    }

    inline const DCoordinate3 DCoordinate3::operator -() const
    {
        return DCoordinate3(-_data[0], -_data[1], -_data[2]);
    }

    // add
    inline const DCoordinate3 DCoordinate3::operator +(const DCoordinate3& rhs) const
    {
        return DCoordinate3(_data[0] + rhs._data[0], _data[1] + rhs._data[1], _data[2] + rhs._data[2]);
    }

    // add to *this
    inline DCoordinate3& DCoordinate3::operator +=(const DCoordinate3& rhs)
    {
        _data[0] += rhs._data[0];
        _data[1] += rhs._data[1];
        _data[2] += rhs._data[2];
        return *this;
    }

    // subtract
    inline const DCoordinate3 DCoordinate3::operator -(const DCoordinate3& rhs) const
    {
        // homework
    }

    // subtract from *this
    inline DCoordinate3& DCoordinate3::operator -=(const DCoordinate3& rhs)
    {
        //homework
    }
```

Descartes coordinates – header file, part VI

DCoordinates3.h

```
    // cross product
    inline const DCoordinate3 DCoordinate3::operator ^(const DCoordinate3& rhs) const
    {
        return DCoordinate3(
                _data[1] * rhs._data[2] - _data[2] * rhs._data[1],
                _data[2] * rhs._data[0] - _data[0] * rhs._data[2],
                _data[0] * rhs._data[1] - _data[1] * rhs._data[0]);
    }

    // cross product, result is stored by *this
    inline DCoordinate3& DCoordinate3::operator ^=(const DCoordinate3& rhs)
    {
        // homework
    }

    // dot product
    inline const GLdouble DCoordinate3::operator *(const DCoordinate3& rhs) const
    {
        return _data[0] * rhs._data[0] + _data[1] * rhs._data[1] + _data[2] * rhs._data[2];
    }

    // scale
    inline const DCoordinate3 DCoordinate3::operator *(const GLdouble& rhs) const
    {
        return DCoordinate3(_data[0] * rhs, _data[1] * rhs, _data[2] * rhs);
    }

    inline const DCoordinate3 operator *(const GLdouble& lhs, const DCoordinate3& rhs)
    {
        // homework
    }

    inline const DCoordinate3 DCoordinate3::operator /(const GLdouble& rhs) const
    {
        // homework
```

Descartes coordinates – header file, part VII

DCoordinates3.h

```
    }

    // scale *this
    inline DCoordinate3& DCoordinate3::operator *=(const GLdouble& rhs)
    {
        _data[0] *= rhs;
        _data[1] *= rhs;
        _data[2] *= rhs;

        return *this;
    }

    inline DCoordinate3& DCoordinate3::operator /=(const GLdouble& rhs)
    {
        // homework
    }

    // length
    inline const GLdouble DCoordinate3::length() const
    {
        return std::sqrt((*this) * (*this));
    }

    // normalize
    inline DCoordinate3& DCoordinate3::normalize()
    {
        GLdouble l = length();

        if (l && l != 1.0)
            *this /= l;

        return *this;
    }
```

Descartes coordinates – header file, part VIII

DCoordinates3.h

```
    //----------------------------------------------------------------------------
    // definitions of overloaded input/output from/to stream operators
    //----------------------------------------------------------------------------

    // output to stream
    inline std::ostream& operator <<(std::ostream& lhs, const DCoordinate3& rhs)
    {
        return lhs << rhs[0] << "␣" << rhs[1] << "␣" << rhs[2];
    }

    // input from stream
    inline std::istream& operator >>(std::istream& lhs, DCoordinate3& rhs)
    {
        // homework
    }
}
```

Descartes coordinates – source file

∄ DCoordinates3.cpp

Homework

Implement all operators and methods of class DCoordinate3! Notice that for efficiency reasons all operators, methods must be inlined.

A simple template for matrices

Template class Matrix

Description

- By means of the template class Matrix we can represent collocation matrices, control polygons, control nets, grouped spline or patch information.

A simple template for matrices – header file, part I

Matrices.h

```
#pragma once

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

namespace cagd
{
    // forward declaration of template class Matrix
    template <typename T>
    class Matrix;

    // forward declaration of template class RowMatrix
    template <typename T>
    class RowMatrix;

    // forward declaration of template class ColumnMatrix
    template <typename T>
    class ColumnMatrix;

        // forward declaration of template class TriangularMatrix
    template <typename T>
    class TriangularMatrix;

    // forward declarations of overloaded and templated input/output from/to stream operators
    template <typename T>
    std::ostream& operator << (std::ostream& lhs, const Matrix<T>& rhs);

    template <typename T>
    std::istream& operator >>(std::istream& lhs, Matrix<T>& rhs);

    template <typename T>
    std::istream& operator >>(std::istream& lhs, TriangularMatrix<T>& rhs);

```

A simple template for matrices – header file, part II

Matrices.h

```
template <typename T>
std::ostream& operator << (std::ostream& lhs, const TriangularMatrix<T>& rhs);

//----------------------------------------------------------
// template class Matrix
//----------------------------------------------------------
template <typename T>
class Matrix
{
    friend std::ostream& operator << <T>(std::ostream&, const Matrix<T>& rhs);
    friend std::istream& operator >> <T>(std::istream&, Matrix<T>& rhs);

protected:
    GLuint                              _row_count;
    GLuint                              _column_count;
    std::vector< std::vector<T> >       _data;
public:
    // special constructor (can also be used as a default constructor)
    Matrix(GLuint row_count = 1, GLuint column_count = 1);

    // copy constructor
    Matrix(const Matrix& m);

    // assignment operator
    Matrix& operator =(const Matrix& m);

    // get element by reference
    T& operator ()(GLuint row, GLuint column);

    // get copy of an element
    T operator ()(GLuint row, GLuint column) const;

    // get dimensions
    GLuint GetRowCount() const;
    GLuint GetColumnCount() const;
```

A simple template for matrices – header file, part III

Matrices.h

```
        // set dimensions
        virtual GLboolean ResizeRows(GLuint row_count);
        virtual GLboolean ResizeColumns(GLuint column_count);

        // update
        GLboolean SetRow(GLuint index, const RowMatrix<T>& row);
        GLboolean SetColumn(GLuint index, const ColumnMatrix<T>& column);

        // destructor
        virtual ~Matrix();
    };

    //--------------------------------
    // template class RowMatrix
    //--------------------------------
    template <typename T>
    class RowMatrix: public Matrix<T>
    {
    public:
        // special constructor (can also be used as a default constructor)
        RowMatrix(GLuint column_count = 1);

        // get element by reference
        T& operator ()(GLuint column);
        T& operator [](GLuint column);

        // get copy of an element
        T operator ()(GLuint column) const;
        T operator [](GLuint column) const;

        // a row matrix consists of a single row
        GLboolean ResizeRows(GLuint row_count);
    };
```

A simple template for matrices – header file, part IV

Matrices.h

```cpp
//----------------------------------
// template class ColumnMatrix
//----------------------------------
template <typename T>
class ColumnMatrix: public Matrix<T>
{
public:
    // special constructor (can also be used as a default constructor)
    ColumnMatrix(GLuint row_count = 1);

    // get element by reference
    T& operator ()(GLuint row);
    T& operator [](GLuint row);

    // get copy of an element
    T operator ()(GLuint row) const;
    T operator [](GLuint row) const;

    // a column matrix consists of a single column
    GLboolean ResizeColumns(GLuint column_count);
};

//--------------------------------------
// template class TriangularMatrix
//--------------------------------------
template <typename T>
class TriangularMatrix
{
    friend std::istream& operator >> <T>(std::istream&, TriangularMatrix<T>& rhs);
    friend std::ostream& operator << <T>(std::ostream&, const TriangularMatrix<T>& rhs);

protected:
    GLuint                       _row_count;
    std::vector< std::vector<T> > _data;
```

A simple template for matrices – header file, part V

Matrices.h

```
    public:
        // special constructor (can also be used as a default constructor)
        TriangularMatrix(GLuint row_count = 1);

        // get element by reference
        T& operator ()(GLuint row, GLuint column);

        // get copy of an element
        T operator ()(GLuint row, GLuint column) const;

        // get dimension
        GLuint GetRowCount() const;

        // set dimension
        GLboolean ResizeRows(GLuint row_count);
    };

    //----------------------------------------------------------
    // homework: implementation of template class Matrix
    //----------------------------------------------------------

    //-------------------------------------------------------------
    // homework: implementation of template class RowMatrix
    //-------------------------------------------------------------


    //----------------------------------------------------------------
    // homework: implementation of template class ColumnMatrix
    //----------------------------------------------------------------

    //--------------------------------------------------------------------
    // homework: implementation of template class TriangularMatrix
    //--------------------------------------------------------------------
```

A simple template for matrices – header file, part VI

Matrices.h

```
    //--------------------------------------------------------------------------------------------------
    // definitions of Matrix-related overloaded and templated input/output from/to stream operators
    //--------------------------------------------------------------------------------------------------

    // output to stream
    template <typename T>
    std::ostream& operator <<(std::ostream& lhs, const Matrix<T>& rhs)
    {
        lhs << rhs._row_count << "␣" << rhs._column_count << std::endl;
        for (typename std::vector< std::vector<T> >::const_iterator row = rhs._data.begin();
             row != rhs._data.end(); ++row)
        {
            for (typename std::vector<T>::const_iterator column = row->begin();
                 column != row->end(); ++column)
                    lhs << *column << "␣";
            lhs << std::endl;
        }
        return lhs;
    }

    // input from stream
    template <typename T>
    std::istream& operator >>(std::istream& lhs, Matrix<T>& rhs)
    {
        // homework
    }

    //--------------------------------------------------------------------------------------------
    // definitions of TringularMatrix-related overloaded and templated input/output from/to
    // stream operators
    //--------------------------------------------------------------------------------------------

    // homework
}
```

A simple template for matrices – source file

∄ Matrices.cpp

Homework

Implement all operators, methods and friend functions of the template classes Matrix, RowMatrix, and ColumnMatrix! The implementation must be done in the header file Matrices.h.

Real square matrices

Derived class RealSquareMatrix

Description

- Matrix<GLdouble> is the base class of the class RealSquareMatrix.
- Some numerical methods (e.g. data point interpolation, degree elevation, LU-decompisition, solutions of linear systems) require real square matrices.

Real square matrices – header file, part I

RealSquareMatrices.h

```
#pragma once

#include "DCoordinates3.h"
#include <GL/glew.h>
#include <limits>
#include "Matrices.h"

namespace cagd
{
    class RealSquareMatrix: public Matrix<GLdouble>
    {
    private:
        GLboolean           _lu_decomposition_is_done;
        std::vector<GLuint> _row_permutation;

    public:
        // special constructor
        RealSquareMatrix(GLuint size);

        // homework: copy constructor
        RealSquareMatrix(const RealSquareMatrix& m);

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

        // homework: square matrices have the same number of rows and columns!
        GLboolean ResizeRows(GLuint row_count);
        GLboolean ResizeColumns(GLuint row_count);

        // tries to determine the LU decomposition of this square matrix
        GLboolean PerformLUDecomposition();

        // Solves linear systems of type A * x = b, where A is a regular square matrix,
        // while b and x are row or column matrices with elements of type T.
        // Here matrix A corresponds to *this.
```

Real square matrices – header file, part II

RealSquareMatrices.h

```cpp
        // Advantage: T can be either GLdouble or DCoordinate,
        // or any other type which has similar mathematical operators.
        template <class T>
        GLboolean SolveLinearSystem(const Matrix<T>& b, Matrix<T>& x,
                                    GLboolean represent_solutions_as_columns = GL_TRUE);
    };

    template <class T>
    GLboolean RealSquareMatrix::SolveLinearSystem(const Matrix<T>& b, Matrix<T>& x,
                                                  GLboolean represent_solutions_as_columns)
    {
        if (!_lu_decomposition_is_done)
            if (!PerformLUDecomposition())
                return GL_FALSE;

        if (represent_solutions_as_columns)
        {
            GLint size = static_cast<GLint>(GetColumnCount());
            if (static_cast<GLint>(b.GetRowCount()) != size)
                    return GL_FALSE;

            x = b;

            for (GLuint k = 0; k < b.GetColumnCount(); ++k)
            {
                GLint ii = 0;
                for (GLint i = 0; i < size; ++i)
                {
                    GLuint ip = _row_permutation[i];
                    T sum = x(ip, k);
                    x(ip, k) = x(i, k);
                    if (ii != 0)
                        for (GLint j = ii - 1; j < i; ++j)
                            sum -= _data[i][j] * x(j, k);
                    else
```

Real square matrices – header file, part III

RealSquareMatrices.h

```
                        if (sum != 0.0)
                            ii = i + 1;
                    x(i, k) = sum;
                }

                for (GLint i = size - 1; i >= 0; --i)
                {
                    T sum = x(i, k);
                    for (GLint j = i + 1; j < size; ++j)
                        sum -= _data[i][j] * x(j, k);
                    x(i, k) = sum /= _data[i][i];
                }
            }
        }
        else
        {
            GLint size = static_cast<GLint>(GetRowCount());
            if (static_cast<GLint>(b.GetColumnCount()) != size)
                return GL_FALSE;

            x = b;

            for (GLuint k = 0; k < b.GetRowCount(); ++k)
            {
                GLint ii = 0;
                for (GLint i = 0; i < size; ++i)
                {
                    GLuint ip = _row_permutation[i];
                    T sum = x(k, ip);
                    x(k, ip) = x(k, i);
                    if (ii != 0)
                        for (GLint j = ii - 1; j < i; ++j)
                            sum -= _data[i][j] * x(k, j);
                    else
                        if (sum != 0.0)
```

Real square matrices – header file, part IV

RealSquareMatrices.h

```
                            ii = i + 1;
                    x(k, i) = sum;
                }

                for (GLint i = size - 1; i >= 0; --i)
                {
                    T sum = x(k, i);
                    for (GLint j = i + 1; j < size; ++j)
                        sum -= _data[i][j] * x(k, j);
                    x(k, i) = sum /= _data[i][i];
                }
            }
        }

        return GL_TRUE;
    }

}
```

Real square matrices – source file, part I

RealSquareMatrix.cpp

```
#include "RealSquareMatrices.h"

using namespace cagd;
using namespace std;

RealSquareMatrix::RealSquareMatrix(GLuint size):
        Matrix<GLdouble>(size, size),
        _lu_decomposition_is_done(GL_FALSE)
{
}

GLboolean RealSquareMatrix::PerformLUDecomposition()
{
    if (_lu_decomposition_is_done)
        return GL_TRUE;

    if (_row_count <= 1)
        return GL_FALSE;

    const GLdouble tiny = numeric_limits<GLdouble>::min();

    GLuint size = static_cast<GLuint>(_data.size());
    vector<GLdouble> implicit_scaling_of_each_row(size);

    _row_permutation.resize(size);

    GLdouble row_interchanges = 1.0;

    //------------------------------------------------------------
    // loop over rows to get the implicit scaling information
    //------------------------------------------------------------
    vector<GLdouble>::iterator its = implicit_scaling_of_each_row.begin();
    for (vector<vector<GLdouble> >::const_iterator itr = _data.begin(); itr < _data.end(); ++itr)
    {
        GLdouble big = 0.0;
```

Real square matrices – source file, part II

RealSquareMatrix.cpp

```
        for (vector<GLdouble>::const_iterator itc = itr->begin(); itc < itr->end(); ++itc)
        {
            GLdouble temp = abs(*itc);
            if (temp > big)
                    big = temp;
        }

        if (big == 0.0)
        {
            // the matrix is singular
            return GL_FALSE;
        }
        *its = 1.0 / big;
        ++its;
    }

    //-----------------------------------
    // search for the largest pivot element
    //-----------------------------------
    for (GLuint k = 0; k < size; ++k)
    {
        GLuint imax = k;
        GLdouble big = 0.0;
        for (GLuint i = k; i < size; ++i)
        {
            GLdouble temp = implicit_scaling_of_each_row[i] * abs(_data[i][k]);
            if (temp > big)
            {
                big = temp;
                imax = i;
            }
        }

        // do we need to interchange rows?
        if (k != imax)
```

Real square matrices – source file, part III

RealSquareMatrix.cpp

```
        {
            for (GLuint j = 0; j < size; ++j)
            {
                GLdouble temp = _data[imax][j];
                _data[imax][j] = _data[k][j];
                _data[k][j] = temp;
            }
            // change the parity of row_interchanges
            row_interchanges = -row_interchanges;
            // also interchange the scale factor
            implicit_scaling_of_each_row[imax] = implicit_scaling_of_each_row[k];
        }

        _row_permutation[k] = imax;
        if (_data[k][k] == 0.0)
            _data[k][k] = tiny;

        for (GLuint i = k + 1; i < size; ++i)
        {
            // divide by pivot element
            GLdouble temp = _data[i][k] /= _data[k][k];

            // reduce remaining submatrix
            for (GLuint j = k + 1; j < size; ++j)
                _data[i][j] -= temp * _data[k][j];
        }
    }

    _lu_decomposition_is_done = GL_TRUE;

    return GL_TRUE;
}
```

Real square matrices – source file, part IV

RealSquareMatrix.cpp

Homework

Implement all unfinished methods and operators of the class RealSquareMatrix!

Generic curves

Class GenericCurve3

Description

- Class GenericCurve3 can also be used as a base class for any type of curve.
- This class provides methods only for rendering and updating (i.e. does not implement coordinate or blending functions).
- When using inheritance, the coordinates of the curve points, first and second order derivatives must be set either by one of the methods (e.g. constructor) of the derived class, or via the inherited methods

```
DCoordinate3& operator ()(GLuint order, GLuint index);

GLboolean SetDerivative(GLuint order, GLuint index, GLdouble x, GLdouble y, GLdouble z = 0.0);
GLboolean SetDerivative(GLuint order, GLuint index, const DCoordinate3 &d);
```

- The rendering is based on vertex buffer objects. Notice that, for efficiency reasons all double coordinates are truncated to float numbers when creating/loading the data of vertex buffer objects.

Generic curves – header file, part I

GenericCurves3.h

```cpp
#pragma once

#include "DCoordinates3.h"
#include <GL/glew.h>
#include "Matrices.h"
#include <iostream>

namespace cagd
{
    //----------------------------------
    // class GenericCurve3
    //----------------------------------
    class GenericCurve3
    {
        //----------------------------------------------
        // input/output from/to stream
        //----------------------------------------------
        friend std::ostream& operator <<(std::ostream& lhs, const GenericCurve3& rhs);
        friend std::istream& operator >>(std::istream& lhs, GenericCurve3& rhs);

    protected:
        GLenum               _usage_flag;
        RowMatrix<GLuint>    _vbo_derivative;
        Matrix<DCoordinate3> _derivative;

    public:
        // default and special constructor
        GenericCurve3(
                GLuint maximum_order_of_derivatives = 2,
                GLuint point_count = 0,
                GLenum usage_flag = GL_STATIC_DRAW);

        // special constructor
        GenericCurve3(const Matrix<DCoordinate3>& derivative, GLenum usage_flag = GL_STATIC_DRAW);
```

Generic curves – header file, part II

GenericCurves3.h

```
        // copy constructor
        GenericCurve3(const GenericCurve3& curve);

        // assignment operator
        GenericCurve3& operator =(const GenericCurve3& rhs);

        // vertex buffer object handling methods
        GLvoid DeleteVertexBufferObjects();
        GLboolean RenderDerivatives(GLuint order, GLenum render_mode) const;
        GLboolean UpdateVertexBufferObjects(GLenum usage_flag = GL_STATIC_DRAW);

        GLfloat* MapDerivatives(GLuint order, GLenum access_mode = GL_READ_ONLY) const;
        GLboolean UnmapDerivatives(GLuint order) const;

        // get derivative by value
        DCoordinate3 operator ()(GLuint order, GLuint index) const;

        // get derivative by reference
        DCoordinate3& operator ()(GLuint order, GLuint index);

        // other update and query methods
        GLboolean SetDerivative(GLuint order, GLuint index, GLdouble x, GLdouble y, GLdouble z = 0.0);
        GLboolean SetDerivative(GLuint order, GLuint index, const DCoordinate3& d);
        GLboolean GetDerivative(GLuint order, GLuint index, GLdouble& x, GLdouble& y, GLdouble& z) const;
        GLboolean GetDerivative(GLuint order, GLuint index, DCoordinate3& d) const;

        GLuint GetMaximumOrderOfDerivatives() const;
        GLuint GetPointCount() const;
        GLenum GetUsageFlag() const;

        // destructor
        virtual ~GenericCurve3();
    };
}
```

Generic curves – source file, part I

GenericCurves3.cpp

```cpp
#include "GenericCurves3.h"

using namespace cagd;
using namespace std;

//--------------------------------------
// implementation of class GenericCurve3
//--------------------------------------

// default and special constructor
GenericCurve3::GenericCurve3(GLuint maximum_order_of_derivatives, GLuint point_count, GLenum usage_flag):
        _usage_flag(usage_flag),
        _vbo_derivative(maximum_order_of_derivatives + 1),
        _derivative(maximum_order_of_derivatives + 1, point_count)
{
}

// special constructor
GenericCurve3::GenericCurve3(const Matrix<DCoordinate3>& derivative, GLenum usage_flag):
        _usage_flag(usage_flag),
        _vbo_derivative(RowMatrix<GLuint>(derivative.GetRowCount())),
        _derivative(derivative)
{
}

// copy constructor
GenericCurve3::GenericCurve3(const GenericCurve3& curve):
        _usage_flag(curve._usage_flag),
        _vbo_derivative(RowMatrix<GLuint>(curve._vbo_derivative.GetColumnCount())),
        _derivative(curve._derivative)
{
    GLboolean vbo_update_is_possible = GL_TRUE;
    for (GLuint i = 0; i < curve._vbo_derivative.GetColumnCount(); ++i)
        vbo_update_is_possible &= curve._vbo_derivative(i);
```

Generic curves – source file, part II

GenericCurves3.cpp

```
    if (vbo_update_is_possible)
        UpdateVertexBufferObjects(_usage_flag);
}

// assignment operator
GenericCurve3& GenericCurve3::operator =(const GenericCurve3& rhs)
{
    if (this != &rhs)
    {
        DeleteVertexBufferObjects();

        _usage_flag = rhs._usage_flag;
        _derivative = rhs._derivative;

        GLboolean vbo_update_is_possible = GL_TRUE;
        for (GLuint i = 0; i < rhs._vbo_derivative.GetColumnCount(); ++i)
            vbo_update_is_possible &= rhs._vbo_derivative(i);

        if (vbo_update_is_possible)
            UpdateVertexBufferObjects(_usage_flag);
    }
    return *this;
}

// vertex buffer object handling methods
GLvoid GenericCurve3::DeleteVertexBufferObjects()
{
    for (GLuint i = 0; i < _vbo_derivative.GetColumnCount(); ++i)
    {
        if (_vbo_derivative(i))
        {
            glDeleteBuffers(1, &_vbo_derivative(i));
            _vbo_derivative(i) = 0;
        }
    }
```

Generic curves – source file, part III

GenericCurves3.cpp

```
}

GLboolean GenericCurve3::RenderDerivatives(GLuint order, GLenum render_mode) const
{
    GLuint max_order = _derivative.GetRowCount();
    if (order >= max_order || !_vbo_derivative(order))
        return GL_FALSE;

    GLuint point_count = _derivative.GetColumnCount();

    glEnableClientState(GL_VERTEX_ARRAY);
        glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative(order));
            glVertexPointer(3, GL_FLOAT, 0, nullptr);

            if (!order)
            {
                if (render_mode != GL_LINE_STRIP &&
                    render_mode != GL_LINE_LOOP  &&
                    render_mode != GL_POINTS)
                {
                    glBindBuffer(GL_ARRAY_BUFFER, 0);
                    glDisableClientState(GL_VERTEX_ARRAY);
                    return GL_FALSE;
                }

                glDrawArrays(render_mode, 0, point_count);
            }
            else
            {
                if (render_mode != GL_LINES && render_mode != GL_POINTS)
                {
                    glBindBuffer(GL_ARRAY_BUFFER, 0);
                    glDisableClientState(GL_VERTEX_ARRAY);
                    return GL_FALSE;
                }
```

Generic curves – source file, part IV

GenericCurves3.cpp

```
                glDrawArrays(render_mode, 0, 2 * point_count);
            }

        glBindBuffer(GL_ARRAY_BUFFER, 0);
    glDisableClientState(GL_VERTEX_ARRAY);

    return GL_TRUE;
}

GLboolean GenericCurve3::UpdateVertexBufferObjects(GLenum usage_flag)
{
    if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  &&
        usage_flag != GL_STREAM_COPY &&
        usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ &&
        usage_flag != GL_DYNAMIC_COPY &&
        usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  &&
        usage_flag != GL_STATIC_COPY)
        return GL_FALSE;

    DeleteVertexBufferObjects();

    _usage_flag = usage_flag;

    for(GLuint d = 0; d < _vbo_derivative.GetColumnCount(); ++d)
    {
        glGenBuffers(1, &_vbo_derivative(d));

        if (!_vbo_derivative(d))
        {
            for (GLuint i = 0; i < d; ++i)
            {
                glDeleteBuffers(1, &_vbo_derivative(i));
                _vbo_derivative(i) = 0;
            }
```

Generic curves – source file, part V

GenericCurves3.cpp

```
            return GL_FALSE;
        }
    }

    GLuint curve_point_count = _derivative.GetColumnCount();

    GLfloat *coordinate = nullptr;

    // curve points
    GLuint curve_point_byte_size = 3 * curve_point_count * sizeof(GLfloat);

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative(0));
    glBufferData(GL_ARRAY_BUFFER, curve_point_byte_size, nullptr, _usage_flag);

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

    if (!coordinate)
    {
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        DeleteVertexBufferObjects();
        return GL_FALSE;
    }

    for (GLuint i = 0; i < curve_point_count; ++i)
    {
        for (GLuint j = 0; j < 3; ++j)
        {
            *coordinate = static_cast<GLfloat>(_derivative(0,i)[j]);
            ++coordinate;
        }
    }

    if (!glUnmapBuffer(GL_ARRAY_BUFFER))
    {
```

Generic curves – source file, part VI

GenericCurves3.cpp

```
            glBindBuffer(GL_ARRAY_BUFFER, 0);
            DeleteVertexBufferObjects();
            return GL_FALSE;
        }

        // higher order derivatives
        GLuint higher_order_derivative_byte_size = 2 * curve_point_byte_size;

        for (GLuint d = 1; d < _derivative.GetRowCount(); ++d)
        {
            glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative(d));
            glBufferData(GL_ARRAY_BUFFER, higher_order_derivative_byte_size, nullptr, _usage_flag);

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

            if (!coordinate)
            {
                glBindBuffer(GL_ARRAY_BUFFER, 0);
                DeleteVertexBufferObjects();
                return GL_FALSE;
            }

            for (GLuint i = 0; i < curve_point_count; ++i)
            {
                DCoordinate3 sum = _derivative(0, i);
                sum += _derivative(d, i);

                for (GLint j = 0; j < 3; ++j)
                {
                    *coordinate = static_cast<GLfloat>(_derivative(0, i)[j]);
                    *(coordinate + 3) = static_cast<GLfloat>(sum[j]);
                    ++coordinate;
                }

                coordinate += 3;
```

Generic curves – source file, part VII

GenericCurves3.cpp

```cpp
        }

        if (!glUnmapBuffer(GL_ARRAY_BUFFER))
        {
            glBindBuffer(GL_ARRAY_BUFFER, 0);
            DeleteVertexBufferObjects();
            return GL_FALSE;
        }
    }

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    return GL_TRUE;
}

GLfloat* GenericCurve3::MapDerivatives(GLuint order, GLenum access_mode) const
{
    if (order >= _derivative.GetRowCount())
        return 0;

    if (access_mode != GL_READ_ONLY && access_mode != GL_WRITE_ONLY && access_mode != GL_READ_WRITE)
        return 0;

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative(order));

    return (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, access_mode);
}

GLboolean GenericCurve3::UnmapDerivatives(GLuint order) const
{
    if (order >= _derivative.GetRowCount())
        return GL_FALSE;

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative(order));
```

Generic curves – source file, part VIII

GenericCurves3.cpp

```
    return glUnmapBuffer(GL_ARRAY_BUFFER);
}

// get derivative by value
DCoordinate3 GenericCurve3::operator ()(GLuint order, GLuint index) const
{
    return _derivative(order, index);
}

// get derivative by reference
DCoordinate3& GenericCurve3::operator ()(GLuint order, GLuint index)
{
    return _derivative(order, index);
}

// other update and query methods
GLboolean GenericCurve3::SetDerivative(GLuint order, GLuint index, GLdouble x, GLdouble y, GLdouble z)
{
    if (order >= _derivative.GetRowCount() || index >= _derivative.GetColumnCount())
        return GL_FALSE;

    _derivative(order, index)[0] = x;
    _derivative(order, index)[1] = y;
    _derivative(order, index)[2] = z;

    return GL_TRUE;
}

GLboolean GenericCurve3::SetDerivative(GLuint order, GLuint index, const DCoordinate3& d)
{
    if (order >= _derivative.GetRowCount() || index >= _derivative.GetColumnCount())
        return GL_FALSE;

    _derivative(order, index) = d;
```

Generic curves – source file, part IX

GenericCurves3.cpp

```cpp
    return GL_TRUE;
}

GLboolean GenericCurve3::GetDerivative(GLuint order, GLuint index, GLdouble& x, GLdouble& y, GLdouble& z)
{
    if (order >= _derivative.GetRowCount() || index >= _derivative.GetColumnCount())
        return GL_FALSE;

    x = _derivative(order, index)[0];
    y = _derivative(order, index)[1];
    z = _derivative(order, index)[2];

    return GL_TRUE;
}

GLboolean GenericCurve3::GetDerivative(GLuint order, GLuint index, DCoordinate3& d) const
{
    if (order >= _derivative.GetRowCount() || index >= _derivative.GetColumnCount())
        return GL_FALSE;

    d = _derivative(order, index);

    return GL_TRUE;
}

GLuint GenericCurve3::GetMaximumOrderOfDerivatives() const
{
    return _derivative.GetRowCount() - 1;
}

GLuint GenericCurve3::GetPointCount() const
{
    return _derivative.GetColumnCount();
}
```

Generic curves – source file, part X

GenericCurves3.cpp

```cpp
GLenum GenericCurve3::GetUsageFlag() const
{
    return _usage_flag;
}

// destructor
GenericCurve3::~GenericCurve3()
{
    DeleteVertexBufferObjects();
}

//----------------------------------------------------------------
// input/output from/to stream
//----------------------------------------------------------------
ostream& cagd::operator <<(ostream& lhs, const GenericCurve3& rhs)
{
    return lhs << rhs._usage_flag << " " << rhs._derivative << endl;
}

std::istream& cagd::operator >>(std::istream& lhs, GenericCurve3& rhs)
{
    rhs.DeleteVertexBufferObjects();

    return lhs >> rhs._usage_flag >> rhs._derivative;
}
```

Abstract linear combinations

Reminder: An interactive description form of curves

Memento: In CAGD the most widespread description form of curves is the *linear combination*

$$\left\{ \begin{array}{l} \mathbf{c} : [u_{\min}, u_{\max}] \to \mathbb{R}^{\delta},\ \delta \geq 2, \\ \mathbf{c}\,(u) = \displaystyle\sum_{i=0}^{n} \mathbf{p}_i F_{n,i}\,(u) \end{array} \right. \tag{1}$$

of vectors $\mathbf{p}_i \in \mathbb{R}^{\delta}$ and the continuous functions of the system

$$\mathcal{F}_n = \left\{ F_{n,i} : [u_{\min}, u_{\max}] \to \mathbb{R} \right\}_{i=0}^{n}. \tag{2}$$

- In most cases vectors $[\mathbf{p}_i]_{i=0}^{n}$ represent control points forming a control polygon. However, these vectors may correspond to other geometric properties such as tangent and acceleration vectors.
- This means that a curve can be specified by just a few user defined information, which is advantageous from the point of view of data storage and transmission.
- Observe, that curve (1) can be written into the matrix form

$$\mathbf{c}\,(u) = \left[\begin{array}{cccc} \mathbf{p}_0 & \mathbf{p}_1 & \cdots & \mathbf{p}_n \end{array} \right] \left[\begin{array}{c} F_{n,0}\,(u) \\ F_{n,1}\,(u) \\ \vdots \\ F_{n,n}\,(u) \end{array} \right], \forall u \in [u_{\min}, u_{\max}]. \tag{3}$$

Abstract linear combinations

Implementation details: classes LinearCombination3::Derivatives and LinearCombination3

Description

- Class LinearCombination3::Derivatives is a column matrix that stores the r-th ($r \geq 0$) order derivatives of any 2- and 3-dimensional linear combination (spline) at a given knot value.
- The abstract class LinearCombination3 can be used as a base class for any type of curve which is based on an approximation method (like Bézier, NURBS, or cyclic curves). The derived class needs to implement the abstract methods

```
virtual GLboolean BlendingFunctionValues(GLdouble knot,
                      RowMatrix<GLdouble>& values) const = 0;

virtual GLboolean CalculateDerivatives(GLuint max_order_of_derivatives,
                      GLdouble u, Derivatives& data) const = 0;
```

  in order to be able to generate the shape of the curve and to solve the curve interpolation problem.

- It is possible that in some cases the curve interpolation problem can be solved more efficiently than using the method

```
virtual GLboolean UpdateDataForInterpolation(
        const ColumnMatrix<GLdouble>& knot_vector,
        const ColumnMatrix<DCoordinate3>& data_points_to_interpolate);
```

  which is based on LU decomposition. In such cases the derived class can redeclare and implement this virtual method in order to provide a more efficient solution to this problem.

Abstract linear combinations - header file, part I

Implementation details: LinearCombination3.h

```
#pragma once

#include "DCoordinates3.h"
#include "GenericCurves3.h"
#include "Matrices.h"

namespace cagd
{
    //----------------------------------------------
    // class LinearCombination3
    //----------------------------------------------
    class LinearCombination3
    {
    public:
        class Derivatives: public ColumnMatrix<DCoordinate3>
        {
        public:
            // special/default constructor
            Derivatives(GLuint maximum_order_of_derivatives = 2);

            // copy constructor
            Derivatives(const Derivatives& d);

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

            // all inherited Descartes coordinates are set to the null vector
            GLvoid LoadNullVectors();
        };

    protected:
        GLuint                     _vbo_data;
        GLenum                     _data_usage_flag;
        GLdouble                   _u_min, _u_max; // definition domain
        ColumnMatrix<DCoordinate3>  _data; // vectors appearing in the linear combination $\sum_{i=0}^{n} \mathbf{p}_i F_{n,i}(u)$
```

Abstract linear combinations - header file, part II

Implementation details: LinearCombination3.h

```
    public:
        // special constructor
        LinearCombination3(
                GLdouble u_min, GLdouble u_max,
                GLuint data_count,
                GLenum data_usage_flag = GL_STATIC_DRAW);

        // copy constructor
        LinearCombination3(const LinearCombination3& lc);

        // assignment operator
        LinearCombination3& operator =(const LinearCombination3& rhs);

        // vbo handling methods
        virtual GLvoid DeleteVertexBufferObjectsOfData();
        virtual GLboolean RenderData(GLenum render_mode = GL_LINE_STRIP) const;
        virtual GLboolean UpdateVertexBufferObjectsOfData(GLenum usage_flag = GL_STATIC_DRAW);

        // get data by value
        DCoordinate3 operator [](GLuint index) const;

        // get data by reference
        DCoordinate3& operator [](GLuint index);

        // set/get definition domain
        GLvoid SetDefinitionDomain(GLdouble u_min, GLdouble u_max);
        GLvoid GetDefinitionDomain(GLdouble& u_min, GLdouble& u_max) const;

        //------------------------
        // abstract method
        //------------------------
        // calculates a row matrix which consists of function values {F_{n,i}(u)}_{i=0}^{n}
        virtual GLboolean BlendingFunctionValues(GLdouble u, RowMatrix<GLdouble>& values) const = 0;
```

Abstract linear combinations - header file, part III

Implementation details: LinearCombination3.h

```
        //----------------------------------
        // abstract method
        //----------------------------------
        // calculates the point and its associated (higher) order derivatives of the linear
        // combination $\sum_{i=0}^{n} \mathbf{p}_i F_{n,i}(u)$ at the parameter value $u$
        virtual GLboolean CalculateDerivatives(GLuint max_order_of_derivatives, GLdouble u,
                                               Derivatives& d) const = 0;

        // generate image/arc
        virtual GenericCurve3* GenerateImage(GLuint max_order_of_derivatives, GLuint div_point_count,
                                             GLenum usage_flag = GL_STATIC_DRAW) const;

        // assure interpolation
        virtual GLboolean UpdateDataForInterpolation(
                                const ColumnMatrix<GLdouble>& knot_vector,
                                const ColumnMatrix<DCoordinate3>& data_points_to_interpolate);

        // destructor
        virtual ~LinearCombination3();
    };
}
```

Abstract linear combinations - source file, part I

Implementation details: LinearCombination3.cpp

```
#include "LinearCombination3.h"
#include "RealSquareMatrices.h"

using namespace cagd;
using namespace std;

// special/default constructor
LinearCombination3::Derivatives::Derivatives(GLuint maximum_order_of_derivatives):
                ColumnMatrix<DCoordinate3>(maximum_order_of_derivatives + 1)
{
}

// copy constructor
LinearCombination3::Derivatives::Derivatives(const LinearCombination3::Derivatives& d):
                ColumnMatrix<DCoordinate3>(d)
{
}

// assignment operator
LinearCombination3::Derivatives& LinearCombination3::Derivatives::operator =(
                const LinearCombination3::Derivatives& rhs)
{
    if (this != &rhs)
    {
        ColumnMatrix<DCoordinate3>::operator =(rhs);
    }
    return *this;
}

```

Abstract linear combinations - source file, part II

Implementation details: LinearCombination3.cpp

```cpp
// set every derivative to null vector
GLvoid LinearCombination3::Derivatives::LoadNullVectors()
{
    for (GLuint i = 0; i < _data.size(); ++i)
    {
        for (GLuint j = 0; j < 3; ++j)
            _data[i][0][j] = 0.0;
    }
}

// special constructor
LinearCombination3::LinearCombination3(GLdouble u_min, GLdouble u_max, GLuint data_count,
                                       GLenum data_usage_flag):
        _vbo_data(0),
        _data_usage_flag(data_usage_flag),
        _u_min(u_min), _u_max(u_max),
        _data(data_count)
{
}

// copy constructor
LinearCombination3::LinearCombination3(const LinearCombination3 &lc):
        _vbo_data(0),
        _data_usage_flag(lc._data_usage_flag),
        _u_min(lc._u_min), _u_max(lc._u_max),
        _data(lc._data)
{
    if (lc._vbo_data)
        UpdateVertexBufferObjectsOfData(_data_usage_flag);
}

```

Abstract linear combinations - source file, part III

Implementation details: LinearCombination3.cpp

```
// assignment operator
LinearCombination3& LinearCombination3::operator =(const LinearCombination3& rhs)
{
    if (this != &rhs)
    {
        DeleteVertexBufferObjectsOfData();

        _data_usage_flag = rhs._data_usage_flag;
        _u_min = rhs._u_min;
        _u_max = rhs._u_max;
        _data = rhs._data;

        if (rhs._vbo_data)
            UpdateVertexBufferObjectsOfData(_data_usage_flag);
    }

    return *this;
}

// vbo handling methods
GLvoid LinearCombination3::DeleteVertexBufferObjectsOfData()
{
    if (_vbo_data)
    {
        glDeleteBuffers(1, &_vbo_data);
        _vbo_data = 0;
    }
}

GLboolean LinearCombination3::RenderData(GLenum render_mode) const
{
    if (!_vbo_data)
        return GL_FALSE;

```

Abstract linear combinations - source file, part IV

Implementation details: LinearCombination3.cpp

```
    if (render_mode != GL_LINE_STRIP && render_mode != GL_LINE_LOOP && render_mode != GL_POINTS)
        return GL_FALSE;

    glEnableClientState(GL_VERTEX_ARRAY);
        glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
            glVertexPointer(3, GL_FLOAT, 0, nullptr);
            glDrawArrays(render_mode, 0, _data.GetRowCount());
        glBindBuffer(GL_ARRAY_BUFFER, 0);
    glDisableClientState(GL_VERTEX_ARRAY);

    return GL_TRUE;
}

GLboolean LinearCombination3::UpdateVertexBufferObjectsOfData(GLenum usage_flag)
{
    GLuint data_count = _data.GetRowCount();
    if (!data_count)
        return GL_FALSE;

    if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  && usage_flag != GL_STREAM_COPY
     && usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ && usage_flag != GL_DYNAMIC_COPY
     && usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  && usage_flag != GL_STATIC_COPY)
        return GL_FALSE;

    _data_usage_flag = usage_flag;

    DeleteVertexBufferObjectsOfData();

    glGenBuffers(1, &_vbo_data);
    if (!_vbo_data)
        return GL_FALSE;

    glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
    glBufferData(GL_ARRAY_BUFFER, data_count * 3 * sizeof(GLfloat), nullptr, _data_usage_flag);
```

Abstract linear combinations - source file, part V

Implementation details: LinearCombination3.cpp

```cpp
    GLfloat *coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
    if (!coordinate)
    {
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        DeleteVertexBufferObjectsOfData();
        return GL_FALSE;
    }

    for (GLuint i = 0; i < data_count; ++i)
    {
        for (GLuint j = 0; j < 3; ++j)
        {
            *coordinate = static_cast<GLfloat>(_data[i][j]);
            ++coordinate;
        }
    }

    if (!glUnmapBuffer(GL_ARRAY_BUFFER))
    {
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        DeleteVertexBufferObjectsOfData();
        return GL_FALSE;
    }

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    return GL_TRUE;
}

// get data by value
DCoordinate3 LinearCombination3::operator [](GLuint index) const
{
    return _data[index];
}
```

Abstract linear combinations - source file, part VI

Implementation details: LinearCombination3.cpp

```cpp
// get data by reference
DCoordinate3& LinearCombination3::operator [](GLuint index)
{
    return _data[index];
}

// assure interpolation
GLboolean LinearCombination3::UpdateDataForInterpolation(
        const ColumnMatrix<GLdouble>& knot_vector,
        const ColumnMatrix<DCoordinate3>& data_points_to_interpolate)
{
    GLuint data_count = _data.GetRowCount();
    if (data_count != knot_vector.GetRowCount() ||
        data_count != data_points_to_interpolate.GetRowCount())
        return GL_FALSE;

    RealSquareMatrix collocation_matrix(data_count);

    RowMatrix<GLdouble> current_blending_function_values(data_count);
    for (GLuint r = 0; r < knot_vector.GetRowCount(); ++r)
    {
        if (!BlendingFunctionValues(knot_vector(r), current_blending_function_values))
            return GL_FALSE;
        else
            collocation_matrix.SetRow(r, current_blending_function_values);
    }

    return collocation_matrix.SolveLinearSystem(data_points_to_interpolate, _data);
}

// set/get definition domain
GLvoid LinearCombination3::SetDefinitionDomain(GLdouble u_min, GLdouble u_max)
{
    // homework
}
```

Abstract linear combinations - source file, part VII

Implementation details: LinearCombination3.cpp

```cpp
GLvoid LinearCombination3::GetDefinitionDomain(GLdouble& u_min, GLdouble& u_max) const
{
    // homework
}

// generate image/arc
GenericCurve3* LinearCombination3::GenerateImage(
                GLuint max_order_of_derivatives,
                GLuint div_point_count,
                GLenum usage_flag) const
{
    // homework
}


// destructor
LinearCombination3::~LinearCombination3()
{
    DeleteVertexBufferObjectsOfData();
}
```

Definition (Cyclic basis functions)

- The normalized system

$$\mathcal{C}_{2n}=\left\{C_{2n,i}\left(u\right)=c_n\left(1+\cos\left(u-i\lambda_n\right)\right)^n : u\in\left[0,2\pi\right]\right\}_{i=0}^{2n} \tag{4}$$

of cyclic basis functions [Róth et al., 2009] of order n spans the vector space

$$\mathbb{T}_{2n}=\left\langle 1,\cos\left(u\right),\sin\left(u\right),\ldots,\cos\left(nu\right),\sin\left(nu\right) : u\in\left[0,2\pi\right]\right\rangle$$

of trigonometric polynomials of order at most n, where $\lambda_n=\frac{2\pi}{2n+1}$ is a fixed phase change, while the normalizing constant $c_n=\frac{2^n}{(2n+1)\binom{2n}{n}}$ fulfills the recursion

$$\left\{\begin{array}{lcl} c_1 & = & \dfrac{1}{3}, \\ c_n & = & \dfrac{n}{2n+1}c_{n-1},\ n\geq 2. \end{array}\right. \tag{5}$$

A case study

Cyclic basis functions

- Observe, that the common prime period of basis functions (4) is 2π.
- This basis fulfills the cyclic variation diminishing property and it is useful for smooth closed curve modeling as it is proven in [Róth et al., 2009].
- Fig. 1 presents periodic cyclic basis functions of order 3 (degree 6).

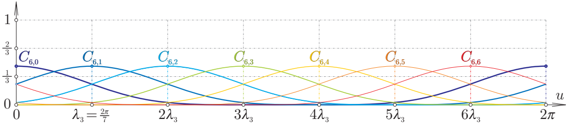

Fig. 1: Cyclic basis functions of order 3.

Definition (Cyclic curves)

The convex combination

$$
\begin{aligned}
\mathbf{c}_n(u) &= \sum_{i=0}^{2n} \mathbf{p}_i C_{2n,i}(u) \\
&= \sum_{i=0}^{2n} \mathbf{p}_i c_n \left(1+\cos\left(u - i\lambda_n\right)\right)^n \\
&= \frac{1}{2n+1} \sum_{i=0}^{2n} \mathbf{p}_i + \frac{2}{(2n+1)\binom{2n}{n}} \sum_{i=0}^{2n} \left(\sum_{k=0}^{n-1} \binom{2n}{k} \cos\left(\left(n-k\right)\left(u - i\lambda_n\right)\right) \right) \mathbf{p}_i, \\
& u \in [0, 2\pi]
\end{aligned}
$$

defines the cyclic curve of order n.

Implementation details: header file, part – I

Cyclic/CyclicCurves3.h

```
#pragma once

#include "../Core/LinearCombination3.h"
#include "../Core/Matrices.h"

namespace cagd
{
    class CyclicCurve3: public LinearCombination3
    {
    protected:
        GLuint                     _n;         // order
        GLdouble                   _c_n;       // normalizing constant
        GLdouble                   _lambda_n;  // phase change

        TriangularMatrix<GLdouble> _bc;        // binomial coefficients

        GLdouble _CalculateNormalizingCoefficient(GLuint n);

        GLvoid   _CalculateBinomialCoefficients(GLuint m, TriangularMatrix<GLdouble> &bc);

    public:
        // special constructor
        CyclicCurve3(GLuint n, GLenum data_usage_flag = GL_STATIC_DRAW);

        // redeclare and define inherited pure virtual methods
        GLboolean BlendingFunctionValues(GLdouble u, RowMatrix<GLdouble> &values) const;
        GLboolean CalculateDerivatives(
                    GLuint max_order_of_derivatives, GLdouble u, Derivatives &d) const;
    };
}
```

Implementation details: source file, part – I

Cyclic/CyclicCurves3.cpp

```
#include "CyclicCurves3.h"

#include "../Core/Constants.h"

#include <iostream>
#include <cmath>

using namespace std;

namespace cagd
{
    GLdouble CyclicCurve3::_CalculateNormalizingCoefficient(GLuint n)
    {
        if (!n)
        {
            return 1.0;
        }

        GLdouble c = 1.0 / 3.0;

        for (GLuint i = 2; i <= n; ++i)
        {
            c *= static_cast<GLdouble>(i) / static_cast<GLdouble>(2 * i + 1);
        }

        return c;
    }

    GLvoid CyclicCurve3::_CalculateBinomialCoefficients(GLuint m, TriangularMatrix<GLdouble> &bc)
    {
        bc.ResizeRows(m + 1);

        bc(0, 0) = 1.0;

```

Implementation details: source file, part – II

Cyclic/CyclicCurves3.cpp

```cpp
        for (GLuint r = 1; r <= m; ++r)
        {
            bc(r, 0) = 1.0;
            bc(r, r) = 1.0;

            for (GLuint i = 1; i <= r / 2; ++i)
            {
                bc(r, i) = bc(r-1, i - 1) + bc(r - 1, i);
                bc(r, r - i) = bc(r, i);
            }
        }
    }

    CyclicCurve3::CyclicCurve3(GLuint n, GLenum data_usage_flag):
            LinearCombination3(0.0, TWO_PI, 2 * n + 1, data_usage_flag),
            _n(n),
            _c_n(_CalculateNormalizingCoefficient(n)),
            _lambda_n(TWO_PI / (2 * n + 1))
    {
        _CalculateBinomialCoefficients(2 * _n, _bc);
    }

    // $\mathcal{C}_{2n} = \left\{C_{2n,i}\left(u\right) = c_n\left(1 + \cos\left(u - i\lambda_n\right)\right)^n : u \in \left[0, 2\pi\right]\right\}_{i=0}^{2n}$
    GLboolean CyclicCurve3::BlendingFunctionValues(GLdouble u, RowMatrix<GLdouble>& values) const
    {
        values.ResizeColumns(2 * _n + 1);

        for (GLuint i = 0; i <= 2 * _n; ++i)
        {
            values[i] = _c_n * pow(1.0 + cos(u - i * _lambda_n), static_cast<GLint>(_n));
        }

        return GL_TRUE;
    }
```

Implementation details: source file, part – III

Cyclic/CyclicCurves3.cpp

```
// $\mathbf{c}_n(u) = \frac{1}{2n+1}\sum_{i=0}^{2n}\mathbf{p}_i + \frac{2}{(2n+1)\binom{2n}{n}}\sum_{i=0}^{2n}\left(\sum_{k=0}^{n-1}\binom{2n}{k}\cos((n-k)(u-i\lambda_n))\right)\mathbf{p}_i,$
// $\frac{\mathrm{d}^r}{\mathrm{d}u^r}\mathbf{c}_n(u) = \frac{2}{(2n+1)\binom{2n}{n}}\sum_{i=0}^{2n}\left(\sum_{k=0}^{n-1}(n-k)^r\binom{2n}{k}\cos\left((n-k)(u-i\lambda_n)+\frac{r\pi}{2}\right)\right)\mathbf{p}_i,\ r \geq 1$
GLboolean CyclicCurve3::CalculateDerivatives(
            GLuint max_order_of_derivatives, GLdouble u, Derivatives& d) const
{
    d.ResizeRows(max_order_of_derivatives + 1);
    d.LoadNullVectors();

    DCoordinate3 centroid;

    for (GLuint i = 0; i <= 2 * _n; ++i)
    {
        centroid += _data[i];
    }
    centroid /= static_cast<GLdouble>(2 * _n + 1);

    for (GLuint r = 0; r <= max_order_of_derivatives; ++r)
    {
        for (GLuint i = 0; i <= 2 * _n; ++i)
        {
            GLdouble sum_k = 0.0;

            for (GLuint k = 0; k <= _n - 1; ++k)
            {
                sum_k += pow(_n - k, static_cast<GLint>(r)) *
                         _bc(2 * _n, k) *
                         cos((_n - k) * (u - i * _lambda_n) + r * PI / 2.0);
            }
```

Implementation details: source file, part – IV

Cyclic/CyclicCurves3.cpp

```
                d[r] += sum_k * _data[i];
            }

            d[r] *= 2.0;
            d[r] /= static_cast<GLdouble>(2 * _n + 1);
            d[r] /= _bc(2 * _n, _n);
        }

        d[0] += centroid;

        return GL_TRUE;
    }
}
```

Bibliography

Róth, Á., Juhász, I., Schicho, J., Hoffmann, M., **2009**.
A cyclic basis for closed curve and surface modeling,
Computer Aided Geometric Design, **26**(5):528–546,
`https://doi.org/10.1016/j.cagd.2009.02.002`.