```
// Copyright (C) 2002-2014 Benjamin Hampe
// This file is part of the "irrlicht-engine"
// For conditions of distribution and use, see copyright notice in irrlicht.h

#ifndef __IRR_EXT_C_DYNAMIC_RECTANGLE_MATRIX_H__
#define __IRR_EXT_C_DYNAMIC_RECTANGLE_MATRIX_H__

#include <irrTypes.h>
#include <irrMath.h>
#include <irrString.h>

namespace irr
{
namespace core
{
    /// @class Dynamic rectangular ( m x n ) matrix as template
    template <class ElementType>
    class CMatrix : public IReferenceCounted
    {
    private:
        /// @brief Pointer to data
        ElementType** Data;

        /// @brief Number of rows ( y-direction )
        u32 Rows;

        /// @brief Number of columns ( x-direction )
        u32 Cols;

        /// @brief Name of the matrix
        core::stringc Name;

        /// @brief Linear algebra stuff ( not needed now )
        bool IsIdentity;

        /// @brief Linear algebra stuff ( not needed now )
        bool IsDeterminantDirty;
        ElementType Determinant;

        /// @brief Linear algebra stuff ( not needed now )
        /** Rank is always lower or equal then min of (Rows,Cols).
            It is the number of linear independant basevectors. */
        u32 Rank;

    public:

        /// @brief Create a two-dimensional array using C++ new operator
        /** Warning: Data is still uninitialized after creation.
            Use fill() or similar to get a valid/known state */
        static ElementType** create2DArray(u32 rows, u32 cols)
        {
            dbPRINT( "create2DArray(%d,%d)\n", rows, cols);

            // abort condition
            if ((rows == 0) || (cols == 0))
                return (ElementType**)0;

            // fill with zeros with memset
            const u32 byte_count = (u32)(sizeof(ElementType)*rows*cols);

            // allocate memory for vector that stores vectors of rows
            ElementType** p = new ElementType*[rows];

            // allocate memory for each row vector
            for (u32 y=0; y<rows; y++)
            {
                p[y]=new ElementType[cols];
            }

            return p;
        }

        /// @brief Check if matrix holds any data or is empty.
        bool empty() const
        {
            if (!Data)
                return true;
            else
                return false;
        }

        /// @brief Deallocate all memory used by this class
        /** Can free up a lot of memory */
        void clear()
        {
            dbPRINT( "CMatrix::clear()\n");

            // delete 2D Array of Floats
            if (!empty())
            {
                // loop rows
                for (u32 i=0; i<Rows; i++)
                {
                    // delete each row
                    ElementType* row = Data[i];
                    if (row)
                    {
                        // delete array
                        delete [] row;

                        Data[i] = 0;
                    }
                }

                // delete array of pointer to arrays
                delete [] Data;
                Data = 0;
            }

            Rows = 0;
            Cols = 0;
        }

        /// @brief Fill all matrix-elements with a given value
        void fill( const ElementType& value )
        {
            if (empty())
            {
                return;
            }

            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    Data[y][x] = value;
                }
            }
        }

        /// @brief Resize the dimension of the matrix
        /** @param keepData If false, matrix is initialized (filled) with zeros
```

```
            @param canShrink Is useless for now */
        bool resize(u32 rows, u32 cols, bool keepData = false, bool canShrink = true)
        {
            dbPRINT( "CMatrix::resize(%d,%d)\n", rows, cols );

            clear();

            Data = create2DArray( rows, cols );
            Rows = rows;
            Cols = cols;

            if (!keepData)
                fill( ElementType(0) );

            return true;
        }

        /// @brief Default contructor
        CMatrix()
            : Data(0), Rows(0), Cols(0), Name("")
        {
            dbPRINT( "CMatrix::CMatrix()\n")
        }

        /// @brief Value contructor
        CMatrix( u32 rows, u32 cols )
            : Data(0), Rows(0), Cols(0), Name("")
        {
            dbPRINT( "CMatrix::CMatrix(%d,%d)\n", rows, cols )
            Data = create2DArray( rows, cols );
            Rows = rows;
            Cols = cols;
        }

        /// @brief Destructor
        ~CMatrix()
        {
            dbPRINT( "destructor()\n" );
            clear();
        }

        /// @brief Copy other matrix into this matrix
        CMatrix& assign( const CMatrix& other )
        {
            dbPRINT( "CMatrix::assign()\n" )

            clear();
            resize( other.getRows(), other.getCols() );

            Rows = other.getRows();
            Cols = other.getCols();
            Size = other.getSize();
            Name = other.getName();

            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    Data[y][x] = other.getElement(y,x);
                }
            }
            return *this;
        }

        /// @brief Copy contructor
```

```cpp
        CMatrix( const CMatrix& other )
            : Data(0), Rows(0), Cols(0), Name("")
        {
            dbPRINT( "CMatrix::CMatrix( CMatrix(%d,%d) )\n", other.getRows(), other.getCols() );
            assign( other );
        }

        /// @brief Clone this matrix
        CMatrix clone() const
        {
            return CMatrix( *this );
        }

        /// @brief Quick typedef
        typedef core::vector2d<ElementType> TRange;

        /// @brief Get the minimum and maximum value inside this matrix
        TRange getMinMax() const
        {
            dbPRINT( "CMatrix::getMinMax()\n" );

            if (!Data)
                return TRange(0,0);

            TRange result( FLT_MAX, FLT_MIN );

            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    const ElementType& value = Data[y][x];
                    if ( result.X > value ) result.X = value;
                    if ( result.Y < value ) result.Y = value;
                }
            }

            dbPRINT( "min = %lf, max = %lf\n", (f64)result.X, (f64)result.Y );

            return result;
        }

        /// @brief Get ( public ) access to raw data pointer
        /** Be careful since this can be dangerous */
        ElementType** getData()
        {
            return Data;
        }

        /// @brief Get matrix dimension ( Width == Cols, Height == Rows )
        /** Be careful since element-access to matrix is by row first
            and then by column [y][x], dont mix Height and Width when doing
            the actual element access! */
        core::dimension2du getDimension() const
        {
            return core::dimension2du( Cols, Rows );
        }

        /// @brief Get number of rows this matrix has ( Y-Direction )
        u32 getRows() const
        {
            return Rows;
        }

        /// @brief Get number of columns this matrix has ( X-Direction )
        u32 getCols() const
        {
            return Cols;
        }

        /// @brief Get number of total elements ( rows x colums )
        /** mostly used for linear memory access ( by index ) */
        u32 getSize() const
        {
            return Rows * Cols;
        }

        /// @brief Get name of this matrix
        core::stringc getName() const
        {
            return Name;
        }

        /// @brief Set name of this matrix
        void setName( const core::stringc& name )
        {
            Name = name;
        }

        /// @brief Print matrix internals to a (multiline) string ( UTF-8 )
        core::stringc toString() const
        {
            core::stringc s("CMatrix<T>("); s+=Rows; s+=","; s+=Cols; s+=",";

            if (Name.size()>0)
            {
                s+=Name;    s+=",";
            }
            s+=") = { \n";

            if (Data)
            {
                for (u32 y=0; y<Rows; y++)
                {
                    s+="\t{\t";
                    for (u32 x=0; x<Cols; x++)
                    {
                        s+= core::floor32( (ElementType)Data[y][x] );
                        if (x<Cols-1)
                        {
                            s+=" ";
                        }
                    }
                    s+="\t}";
                    if (y<Rows-1)
                    {
                        s+=",";
                    }
                    s+="\n";
                }
            }
            s+="};\n";

            return s;
        }

        /// @brief Get a matrix element by coords ( with out-of-bounds check )
        ElementType getElement(u32 row, u32 col) const
        {
            _IRR_DEBUG_BREAK_IF( row >= Rows );
            _IRR_DEBUG_BREAK_IF( col >= Cols );
            if ((row<Rows) && (col<Cols))
```

```cpp
            {
                return Data[row][col];
            }
            else
            {
                return ElementType(0);
            }
        }

        /// @brief Get a matrix element by index ( with out-of-bounds check )
        ElementType getElement(u32 index) const
        {
            _IRR_DEBUG_BREAK_IF( index >= getSize() );
            u32 row = index / Cols;
            u32 col = index - (row * Cols);
            if ((row>=0) && (col>=0) && (row<Rows) && (col<Cols))
            {
                return Data[row][col];
            }
            else
            {
                return ElementType(0);
            }
        }

        /// @brief Set a matrix-element by coords ( with out-of-bounds check )
        bool setElement(u32 row, u32 col, ElementType element)
        {
            _IRR_DEBUG_BREAK_IF( row >= Rows );
            _IRR_DEBUG_BREAK_IF( col >= Cols );
            if ((row<Rows) && (col<Cols))
            {
                Data[row][col] = element;
                return true;
            }

            return false;
        }

        /// @brief Set a matrix-element by index ( with out-of-bounds check )
        bool setElement(u32 index, ElementType element)
        {
            _IRR_DEBUG_BREAK_IF( index >= getSize() );
            u32 row = index / Cols;
            u32 col = index - row * Cols;
            if ((row>=0) && (col>=0) && (row<Rows) && (col<Cols))
            {
                Data[row][col] = element;
                return true;
            }

            return false;
        }

        /// @brief Swap two rows within this matrix
        /** Exchanges the pointers of 2 rows. */
        bool swapRows( u32 row_a, u32 row_b )
        {
            dbPRINT( "CMatrix::swapRows()\n");

            if ( row_a == row_b )
                return false;

            if ( row_a >= Rows )
                return false;

            if ( row_b >= Rows )
                return false;

            /// save value at target position
            ElementType* row = Data[row_a];

            /// overwrite target position with new value
            Data[row_a] = Data[row_b];

            /// overwrite source position with save row-data
            Data[row_b] = row;

            return true;
        }

        /// @brief Shift all rows up or down ( does not work for rows < 0 yet )
        /** Exchanges the pointers and does no element-copying, should be fast. */
        bool shiftRows( s32 rows )
        {
            // dbPRINT( "CMatrix::shiftRows( %d )\n", rows);

            if (rows>0)
            {
                for (u32 y=0; y<Rows; y++)
                {
                    s32 i = ( rows+(s32)y );

                    if (i<0) i += Rows;
                    if (i>=(s32)Rows) i -= (s32)Rows;
                    //%((s32)Rows);
                    u32 k = (u32)i;
        //            k = Rows-1-k;
        //            k = k % Rows;

                    /// save value at target position
                    ElementType* row = Data[y];

                    /// overwrite target position with new value
                    Data[y] = Data[k];

                    /// overwrite source position with save row-data
                    Data[k] = row;
                }
            }
//          else
//          {
//              rows = core::abs_<s32>(rows);
//
//              for (u32 y=0; y<Rows; y++)
//              {
//                  u32 k = ( (u32)rows+y )%Rows ;
//
//                  /// save value at target position
//                  ElementType* row = Data[y];
//
//                  /// overwrite target position with new value
//                  Data[y] = Data[k];
//
//                  /// overwrite source position with save row-data
//                  Data[k] = row;
//              }
//
//          }
            return true;
        }

        bool load( const core::stringc& filename )
        {
            dbPRINT( "CMatrix::load( %s )\n", filename.c_str() );
            return true;
        }

        bool save( const core::stringc& filename ) const
        {
            dbPRINT( "CMatrix::save( %s )\n", filename.c_str() );
            return true;
        }

        /// secure access to value ( with out-of-bounds check )
        const ElementType& operator() (u32 index) const
        {
            return Data[ (index<getSize())?index:0 ];
        }

        /// secure access to value ( with out-of-bounds check )
        ElementType& operator() (u32 index)
        {
            return Data[ (index<getSize())?index:0 ];
        }

        /// secure access to value ( with out-of-bounds check )
        const ElementType& operator() (u32 row, u32 col) const
        {
            u32 index = row*Cols+col;
            return Data[ (index<getSize())?index:0 ];
        }

        /// secure access to value ( with out-of-bounds check )
        ElementType& operator() (u32 row, u32 col)
        {
            u32 index = row*Cols+col;
            return Data[ (index<getSize())?index:0 ];
        }


        /// copy operator overload
        CMatrix& operator= ( const CMatrix& other )
        {
            #ifdef _DEBUG
            dbPRINT( "operator= ()\n" );
            #endif // _DEBUG

            return assign(other);
        }

        /// set row-data ( replace ) with array-values
        template <class T>
        bool setRow( u32 row, const T* data, u32 elem_count, ElementType fillSpace = 0.0f )
        {
            if (!data)
            {
                dbPRINT("fillRow() - ERROR Cant set row of empty CMatrix, return false.\n");
                return false;
            }

            if (row >= Rows)
                return false;

            u32 i_max = core::min_<u32>( elem_count, Cols);

            T* p = const_cast<T*>(data);

            for (u32 i=0; i<i_max; i++)
            {
                if (p)
                {
                    ElementType value = (ElementType)(*p);
                    Data[row][i] = value;
                    p++;
                }
                else
                {
                    break;
                }
            }

            if (i_max < Cols)
            {
                for (u32 i=i_max; i<Cols; i++)
                {
                    Data[row][i] = fillSpace;
                }
            }

            return true;
        }

        /// set row-data ( replace ) with array-values
        template <class T>
        bool setRow( u32 row, const core::array<T>& data, bool bFillBounds = false, ElementType fillSpace = 0.0f )
        {
            if (!Data)
            {
                dbPRINT("fillRow() - ERROR Cant set row of empty CMatrix, return false.\n");
                return false;
            }

            if (row >= Rows)
                return false;

            u32 i_max = core::min_<u32>( data.size(), Cols);

            for (u32 i=0; i<i_max; i++)
            {
                Data[row][i] = (ElementType)data[i];
            }

            if (bFillBounds)
            {
                if (i_max < Cols)
                {
                    for (u32 i=i_max; i<Cols; i++)
                    {
                        Data[row][i] = fillSpace;
                    }
                }
            }
            return true;
        }

        ///@brief Equality operator
        /** Compare this to another CMatrix,
            test for equal row- and col-count first,
            if true, then check element-wise for equality until false */
        bool operator==(const CMatrix& other)
        {
            dbPRINT( "CMatrix::operator== ()\n");

            // abort conditions
            if (( *this == other ) ||
                ( Cols != other.getCols() ) ||
                ( Rows != other.getRows() ))
                return *this;

            // then test element-wise for equality
            for (u32 r=0; r<Rows; r++)
            {
                for (u32 c=0; c<Cols; c++)
                {
                    if ( !core::equals( Data[r][c], other[r][c] ) )
                        return false;
                }
            }

            // if all elements are equal, return true.
            return true;
        }

        /// inequality operator
        bool operator!=(const CMatrix& other)
        {
            return ( *this == other );
        }


        /// translation operator '+'
        CMatrix& operator+ ( const ElementType& value )
        {
            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    Data[y][x] = Data[y][x] + value;
                }
            }

            return *this;
        }

        /// '+' operator overload
        CMatrix& operator+= ( const CMatrix& other )
        {
            // abort conditions
            if (( *this == other ) ||
                ( Cols != other.getCols() ) ||
                ( Rows != other.getRows() ))
                return *this;

            // manipulate this
            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    Data[y][x] = Data[y][x] + other.getElement(y,x);
                }
            }

            return *this;
        }

        /// translation operator '-'
        CMatrix& operator- ( const ElementType& value )
        {
            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    Data[y][x] = Data[y][x] - value;
                }
            }
            return *this;
        }

        /// '-' operator overload
        CMatrix& operator-= ( const CMatrix& other )
        {
            // abort conditions
            if (( *this == other ) ||
                ( Cols != other.getCols() ) ||
                ( Rows != other.getRows() ))
                return *this;

            // manipulate this
            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    Data[y][x] = Data[y][x] - other.getElement(y,x);
                }
            }

            return *this;
        }

        /// scale operator '*'
        CMatrix& operator* ( const ElementType& value )
        {
            for (u32 y=0; y<Rows; y++)
            {
                for (u32 x=0; x<Cols; x++)
                {
                    Data[y][x] = Data[y][x] *value;
                }
            }
            return *this;
        }

        /// scale operator '/'
        CMatrix& operator/ ( const ElementType& value )
        {
            if (!core::equals( value, NullValue ))
            {
                const ElementType value_inv_factor = core::reciprocal( value );

                for (u32 y=0; y<Rows; y++)
                {
                    for (u32 x=0; x<Cols; x++)
                    {
                        Data[y][x] = Data[y][x] * value_inv_factor;
                    }
                }
            }

            return *this;
        }

        video::IImage* createHeightMap( ) const
        {
            dbPRINT( "CMatrix::createHeightMap()\n" );

            core::dimension2du img_size( Cols, Rows );
            video::CImage* img = new video::CImage( video::ECF_A8R8G8B8, img_size);
            if (!img)
                return 0;

            img->fill( 0xffffffff );

            const core::vector2df mm = getMinMax();
            const ElementType height = mm.Y - mm.X;

            for (u32 y = 0; y < core::min_<u32>(Rows, img->getDimension().Height); y++)
            {
                for (u32 x = 0; x < core::min_<u32>(Cols, img->getDimension().Width); x++)
                {
                    ElementType value = Data[y][x];
                    value -= mm.X;
                    value /= height;
                    value = core::clamp<ElementType>( value, 0.0f, 1.0f );
                    video::SColorf color( value, value, value, 1.0f );
                    img->setPixel( x, y, color.toSColor() );
                }
            }

            return img;
        }

        video::IImage* createImage( ) const
        {
            dbPRINT( "CMatrix::createImage()\n" );

            core::dimension2du img_size( Cols, Rows );
            video::CImage* img = new video::CImage( video::ECF_A8R8G8B8, img_size);
            if (!img)
                return 0;

            img->fill( 0xffffffff );

            const core::vector2df mm = getMinMax();
            const ElementType height = mm.Y - mm.X;

            for (u32 y = 0; y < core::min_<u32>(Rows, img->getDimension().Height); y++)
            {
                for (u32 x = 0; x < core::min_<u32>(Cols, img->getDimension().Width); x++)
                {
                    ElementType value = Data[y][x];
                    value -= mm.X;
                    value /= height;
                    value = core::clamp<ElementType>( value, 0.0f, 1.0f );
                    video::SColorf color( value, value, value, 1.0f );
                    img->setPixel( x, y, color.toSColor() );
                }
            }

            return img;
        }

        video::ITexture* createTexture( video::IVideoDriver* driver ) const
        {
            dbPRINT( "CMatrix::createTexture()\n" );

            if (!driver)
                return 0;

            video::IImage* img = createImage();

            video::ITexture* tex = driver->addTexture( "createTexture", img, 0 );

            return tex;
        }


        virtual ElementType det() const
        {
            return ElementType(0);
        }

//      /// @brief Junk
//      /** Exchanges the pointers of 2 rows. */
//      bool shiftRow()
//      {
//  //      dbPRINT( "CMatrix::shiftRow()\n" );
//
//          u32 r = 1;
//          ElementType** b = new ElementType*[Rows];
//
//          if (!b)
//              return false;
//
//          u32 k = 0;
//          for (u32 i = r; i<Rows; i++)
//          {
//              b[i] = Data[k];
//              k++;
//          }
//
//          k = 0;
//          for (u32 i = Rows-r; i<Rows; i++)
//          {
//              b[k] = Data[i];
//              k++;
//          }
//
//          for (u32 i = 0; i<Rows; i++)
//          {
//              Data[i] = b[i];
//          }
//
//          delete [] b;
//
//          return true;
//      }

    };

    typedef CMatrix<f32> CMatrixf;

} // end namespace core
} // end namespace irr

#endif // __IRR_EXT_C_DYNAMIC_RECTANGLE_MATRIX_H__
```