```
// 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++)
            {
```

```
67                  p[y]=new ElementType[cols];
68              }
69
70              return p;
71          }
72
73          /// @brief Check if matrix holds any data or is empty.
74          bool empty() const
75          {
76              if (!Data)
77                  return true;
78              else
79                  return false;
80          }
81
82          /// @brief Deallocate all memory used by this class
83          /** Can free up a lot of memory */
84          void clear()
85          {
86              dbPRINT( "CMatrix::clear()\n");
87
88              // delete 2D Array of Floats
89              if (!empty())
90              {
91                  // loop rows
92                  for (u32 i=0; i<Rows; i++)
93                  {
94                      // delete each row
95                      ElementType* row = Data[i];
96                      if (row)
97                      {
98                          // delete array
99                          delete [] row;
100
101                         Data[i] = 0;
102                     }
103                 }
104
105                 // delete array of pointer to arrays
106                 delete [] Data;
107                 Data = 0;
108             }
109
110             Rows = 0;
111             Cols = 0;
112         }
113
114         /// @brief Fill all matrix-elements with a given value
115         void fill( const ElementType& value )
116         {
117             if (empty())
118             {
119                 return;
120             }
121
122             for (u32 y=0; y<Rows; y++)
123             {
124                 for (u32 x=0; x<Cols; x++)
125                 {
126                     Data[y][x] = value;
127                 }
128             }
129         }
130
131         /// @brief Resize the dimension of the matrix
132         /** @param keepData If false, matrix is initialized (filled) with zeros
```

```
133             @param canShrink Is useless for now */
134         bool resize(u32 rows, u32 cols, bool keepData = false, bool canShrink = true
)
135         {
136             dbPRINT( "CMatrix::resize(%d,%d)\n", rows, cols );
137 
138             clear();
139 
140             Data = create2DArray( rows, cols );
141             Rows = rows;
142             Cols = cols;
143 
144             if (!keepData)
145                 fill( ElementType(0) );
146 
147             return true;
148         }
149 
150         /// @brief Default contructor
151         CMatrix()
152             : Data(0), Rows(0), Cols(0), Name("")
153         {
154             dbPRINT( "CMatrix::CMatrix()\n")
155         }
156 
157         /// @brief Value contructor
158         CMatrix( u32 rows, u32 cols )
159             : Data(0), Rows(0), Cols(0), Name("")
160         {
161             dbPRINT( "CMatrix::CMatrix(%d,%d)\n", rows, cols )
162             Data = create2DArray( rows, cols );
163             Rows = rows;
164             Cols = cols;
165         }
166 
167         /// @brief Destructor
168         ~CMatrix()
169         {
170             dbPRINT( "destructor()\n" );
171             clear();
172         }
173 
174         /// @brief Copy other matrix into this matrix
175         CMatrix& assign( const CMatrix& other )
176         {
177             dbPRINT( "CMatrix::assign()\n" )
178 
179             clear();
180             resize( other.getRows(), other.getCols() );
181 
182             Rows = other.getRows();
183             Cols = other.getCols();
184             Size = other.getSize();
185             Name = other.getName();
186 
187             for (u32 y=0; y<Rows; y++)
188             {
189                 for (u32 x=0; x<Cols; x++)
190                 {
191                     Data[y][x] = other.getElement(y,x);
192                 }
193             }
194             return *this;
195         }
196 
197         /// @brief Copy contructor
```

```
198         CMatrix( const CMatrix& other )
199             : Data(0), Rows(0), Cols(0), Name("")
200         {
201             dbPRINT( "CMatrix::CMatrix( CMatrix(%d,%d) )\n", other.getRows(), other.
getCols() );
202             assign( other );
203         }
204 
205         /// @brief Clone this matrix
206         CMatrix clone() const
207         {
208             return CMatrix( *this );
209         }
210 
211         /// @brief Quick typedef
212         typedef core::vector2d<ElementType> TRange;
213 
214         /// @brief Get the minimum and maximum value inside this matrix
215         TRange getMinMax() const
216         {
217             dbPRINT( "CMatrix::getMinMax()\n" );
218 
219             if (!Data)
220                 return TRange(0,0);
221 
222             TRange result( FLT_MAX, FLT_MIN );
223 
224             for (u32 y=0; y<Rows; y++)
225             {
226                 for (u32 x=0; x<Cols; x++)
227                 {
228                     const ElementType& value = Data[y][x];
229                     if ( result.X > value ) result.X = value;
230                     if ( result.Y < value ) result.Y = value;
231                 }
232             }
233 
234             dbPRINT( "min = %lf, max = %lf\n", (f64)result.X, (f64)result.Y );
235 
236             return result;
237         }
238 
239         /// @brief Get ( public ) access to raw data pointer
240         /** Be careful since this can be dangerous */
241         ElementType** getData()
242         {
243             return Data;
244         }
245 
246         /// @brief Get matrix dimension ( Width == Cols, Height == Rows )
247         /** Be careful since element-access to matrix is by row first
248             and then by column [y][x], dont mix Height and Width when doing
249             the actual element access! */
250         core::dimension2du getDimension() const
251         {
252             return core::dimension2du( Cols, Rows );
253         }
254 
255         /// @brief Get number of rows this matrix has ( Y-Direction )
256         u32 getRows() const
257         {
258             return Rows;
259         }
260 
261         /// @brief Get number of columns this matrix has ( X-Direction )
262         u32 getCols() const
```

```
263         {
264             return Cols;
265         }
266 
267         /// @brief Get number of total elements ( rows x colums )
268         /** mostly used for linear memory access ( by index ) */
269         u32 getSize() const
270         {
271             return Rows * Cols;
272         }
273 
274         /// @brief Get name of this matrix
275         core::stringc getName() const
276         {
277             return Name;
278         }
279 
280         /// @brief Set name of this matrix
281         void setName( const core::stringc& name )
282         {
283             Name = name;
284         }
285 
286         /// @brief Print matrix internals to a (multiline) string ( UTF-8 )
287         core::stringc toString() const
288         {
289             core::stringc s("CMatrix<T>("); s+=Rows; s+=","; s+=Cols; s+=",";
290 
291             if (Name.size()>0)
292             {
293                 s+=Name;    s+=",";
294             }
295             s+=") = { \n";
296 
297             if (Data)
298             {
299                 for (u32 y=0; y<Rows; y++)
300                 {
301                     s+="\t{\t";
302                     for (u32 x=0; x<Cols; x++)
303                     {
304                         s+= core::floor32( (ElementType)Data[y][x] );
305                         if (x<Cols-1)
306                         {
307                             s+=" ";
308                         }
309                     }
310                     s+="\t}";
311                     if (y<Rows-1)
312                     {
313                         s+=",";
314                     }
315                     s+="\n";
316                 }
317             }
318             s+="};\n";
319 
320             return s;
321         }
322 
323         /// @brief Get a matrix element by coords ( with out-of-bounds check )
324         ElementType getElement(u32 row, u32 col) const
325         {
326             _IRR_DEBUG_BREAK_IF( row >= Rows );
327             _IRR_DEBUG_BREAK_IF( col >= Cols );
328             if ((row<Rows) && (col<Cols))
```

```
329             {
330                 return Data[row][col];
331             }
332             else
333             {
334                 return ElementType(0);
335             }
336         }
337 
338         /// @brief Get a matrix element by index ( with out-of-bounds check )
339         ElementType getElement(u32 index) const
340         {
341             _IRR_DEBUG_BREAK_IF( index >= getSize() );
342             u32 row = index / Cols;
343             u32 col = index - (row * Cols);
344             if ((row>=0) && (col>=0) && (row<Rows) && (col<Cols))
345             {
346                 return Data[row][col];
347             }
348             else
349             {
350                 return ElementType(0);
351             }
352         }
353 
354         /// @brief Set a matrix-element by coords ( with out-of-bounds check )
355         bool setElement(u32 row, u32 col, ElementType element)
356         {
357             _IRR_DEBUG_BREAK_IF( row >= Rows );
358             _IRR_DEBUG_BREAK_IF( col >= Cols );
359             if ((row<Rows) && (col<Cols))
360             {
361                 Data[row][col] = element;
362                 return true;
363             }
364 
365             return false;
366         }
367 
368         /// @brief Set a matrix-element by index ( with out-of-bounds check )
369         bool setElement(u32 index, ElementType element)
370         {
371             _IRR_DEBUG_BREAK_IF( index >= getSize() );
372             u32 row = index / Cols;
373             u32 col = index - row * Cols;
374             if ((row>=0) && (col>=0) && (row<Rows) && (col<Cols))
375             {
376                 Data[row][col] = element;
377                 return true;
378             }
379 
380             return false;
381         }
382 
383         /// @brief Swap two rows within this matrix
384         /** Exchanges the pointers of 2 rows. */
385         bool swapRows( u32 row_a, u32 row_b )
386         {
387             dbPRINT( "CMatrix::swapRows()\n");
388 
389             if ( row_a == row_b )
390                 return false;
391 
392             if ( row_a >= Rows )
393                 return false;
394 
```

```
395             if ( row_b >= Rows )
396                 return false;
397
398             /// save value at target position
399             ElementType* row = Data[row_a];
400
401             /// overwrite target position with new value
402             Data[row_a] = Data[row_b];
403
404             /// overwrite source position with save row-data
405             Data[row_b] = row;
406
407             return true;
408         }
409
410         /// @brief Shift all rows up or down ( does not work for rows < 0 yet )
411         /** Exchanges the pointers and does no element-copying, should be fast. */
412         bool shiftRows( s32 rows )
413         {
414             // dbPRINT( "CMatrix::shiftRows( %d )\n", rows);
415
416             if (rows>0)
417             {
418                 for (u32 y=0; y<Rows; y++)
419                 {
420                     s32 i = ( rows+(s32)y );
421
422                     if (i<0) i += Rows;
423                     if (i>=(s32)Rows) i -= (s32)Rows;
424                     //%((s32)Rows);
425                     u32 k = (u32)i;
426         //            k = Rows-1-k;
427         //            k = k % Rows;
428
429                     /// save value at target position
430                     ElementType* row = Data[y];
431
432                     /// overwrite target position with new value
433                     Data[y] = Data[k];
434
435                     /// overwrite source position with save row-data
436                     Data[k] = row;
437                 }
438             }
439 //          else
440 //          {
441 //              rows = core::abs_<s32>(rows);
442 //
443 //              for (u32 y=0; y<Rows; y++)
444 //              {
445 //                  u32 k = ( (u32)rows+y )%Rows ;
446 //
447 //                  /// save value at target position
448 //                  ElementType* row = Data[y];
449 //
450 //                  /// overwrite target position with new value
451 //                  Data[y] = Data[k];
452 //
453 //                  /// overwrite source position with save row-data
454 //                  Data[k] = row;
455 //              }
456 //
457 //          }
458             return true;
459         }
460
```

```
461         bool load( const core::stringc& filename )
462         {
463             dbPRINT( "CMatrix::load( %s )\n", filename.c_str() );
464             return true;
465         }
466 
467         bool save( const core::stringc& filename ) const
468         {
469             dbPRINT( "CMatrix::save( %s )\n", filename.c_str() );
470             return true;
471         }
472 
473         /// secure access to value ( with out-of-bounds check )
474         const ElementType& operator() (u32 index) const
475         {
476             return Data[ (index<getSize())?index:0 ];
477         }
478 
479         /// secure access to value ( with out-of-bounds check )
480         ElementType& operator() (u32 index)
481         {
482             return Data[ (index<getSize())?index:0 ];
483         }
484 
485         /// secure access to value ( with out-of-bounds check )
486         const ElementType& operator() (u32 row, u32 col) const
487         {
488             u32 index = row*Cols+col;
489             return Data[ (index<getSize())?index:0 ];
490         }
491 
492         /// secure access to value ( with out-of-bounds check )
493         ElementType& operator() (u32 row, u32 col)
494         {
495             u32 index = row*Cols+col;
496             return Data[ (index<getSize())?index:0 ];
497         }
498 
499 
500         /// copy operator overload
501         CMatrix& operator= ( const CMatrix& other )
502         {
503             #ifdef _DEBUG
504             dbPRINT( "operator= ()\n" );
505             #endif // _DEBUG
506 
507             return assign(other);
508         }
509 
510         /// set row-data ( replace ) with array-values
511         template <class T>
512         bool setRow( u32 row, const T* data, u32 elem_count, ElementType fillSpace =
0.0f )
513         {
514             if (!data)
515             {
516                 dbPRINT("fillRow() - ERROR Cant set row of empty CMatrix, return
false.\n");
517                 return false;
518             }
519 
520             if (row >= Rows)
521                 return false;
522 
523             u32 i_max = core::min_<u32>( elem_count, Cols);
524 
```

```
525             T* p = const_cast<T*>(data);
526
527             for (u32 i=0; i<i_max; i++)
528             {
529                 if (p)
530                 {
531                     ElementType value = (ElementType)(*p);
532                     Data[row][i] = value;
533                     p++;
534                 }
535                 else
536                 {
537                     break;
538                 }
539             }
540
541             if (i_max < Cols)
542             {
543                 for (u32 i=i_max; i<Cols; i++)
544                 {
545                     Data[row][i] = fillSpace;
546                 }
547             }
548
549             return true;
550         }
551
552         /// set row-data ( replace ) with array-values
553         template <class T>
554         bool setRow( u32 row, const core::array<T>& data, bool bFillBounds = false,
ElementType fillSpace = 0.0f )
555         {
556             if (!Data)
557             {
558                 dbPRINT("fillRow() - ERROR Cant set row of empty CMatrix, return
false.\n");
559                 return false;
560             }
561
562             if (row >= Rows)
563                 return false;
564
565             u32 i_max = core::min_<u32>( data.size(), Cols);
566
567             for (u32 i=0; i<i_max; i++)
568             {
569                 Data[row][i] = (ElementType)data[i];
570             }
571
572             if (bFillBounds)
573             {
574                 if (i_max < Cols)
575                 {
576                     for (u32 i=i_max; i<Cols; i++)
577                     {
578                         Data[row][i] = fillSpace;
579                     }
580                 }
581             }
582             return true;
583         }
584
585         ///@brief Equality operator
586         /** Compare this to another CMatrix,
587             test for equal row- and col-count first,
588             if true, then check element-wise for equality until false */
```

```
589         bool operator==(const CMatrix& other)
590         {
591             dbPRINT( "CMatrix::operator== ()\n");
592 
593             // abort conditions
594             if (( *this == other ) ||
595                 ( Cols != other.getCols() ) ||
596                 ( Rows != other.getRows() ))
597                 return *this;
598 
599             // then test element-wise for equality
600             for (u32 r=0; r<Rows; r++)
601             {
602                 for (u32 c=0; c<Cols; c++)
603                 {
604                     if ( !core::equals( Data[r][c], other[r][c] ) )
605                         return false;
606                 }
607             }
608 
609             // if all elements are equal, return true.
610             return true;
611         }
612 
613         /// inequality operator
614         bool operator!=(const CMatrix& other)
615         {
616             return ( *this == other );
617         }
618 
619 
620         /// translation operator '+'
621         CMatrix& operator+ ( const ElementType& value )
622         {
623             for (u32 y=0; y<Rows; y++)
624             {
625                 for (u32 x=0; x<Cols; x++)
626                 {
627                     Data[y][x] = Data[y][x] + value;
628                 }
629             }
630 
631             return *this;
632         }
633 
634         /// '+' operator overload
635         CMatrix& operator+= ( const CMatrix& other )
636         {
637             // abort conditions
638             if (( *this == other ) ||
639                 ( Cols != other.getCols() ) ||
640                 ( Rows != other.getRows() ))
641                 return *this;
642 
643             // manipulate this
644             for (u32 y=0; y<Rows; y++)
645             {
646                 for (u32 x=0; x<Cols; x++)
647                 {
648                     Data[y][x] = Data[y][x] + other.getElement(y,x);
649                 }
650             }
651 
652             return *this;
653         }
654 
```

```
655         /// translation operator '-'
656         CMatrix& operator- ( const ElementType& value )
657         {
658             for (u32 y=0; y<Rows; y++)
659             {
660                 for (u32 x=0; x<Cols; x++)
661                 {
662                     Data[y][x] = Data[y][x] - value;
663                 }
664             }
665             return *this;
666         }
667 
668         /// '-' operator overload
669         CMatrix& operator-= ( const CMatrix& other )
670         {
671             // abort conditions
672             if (( *this == other ) ||
673                 ( Cols != other.getCols() ) ||
674                 ( Rows != other.getRows() ))
675                 return *this;
676 
677             // manipulate this
678             for (u32 y=0; y<Rows; y++)
679             {
680                 for (u32 x=0; x<Cols; x++)
681                 {
682                     Data[y][x] = Data[y][x] - other.getElement(y,x);
683                 }
684             }
685 
686             return *this;
687         }
688 
689         /// scale operator '*'
690         CMatrix& operator* ( const ElementType& value )
691         {
692             for (u32 y=0; y<Rows; y++)
693             {
694                 for (u32 x=0; x<Cols; x++)
695                 {
696                     Data[y][x] = Data[y][x] *value;
697                 }
698             }
699             return *this;
700         }
701 
702         /// scale operator '/'
703         CMatrix& operator/ ( const ElementType& value )
704         {
705             if (!core::equals( value, NullValue ))
706             {
707                 const ElementType value_inv_factor = core::reciprocal( value );
708 
709                 for (u32 y=0; y<Rows; y++)
710                 {
711                     for (u32 x=0; x<Cols; x++)
712                     {
713                         Data[y][x] = Data[y][x] * value_inv_factor;
714                     }
715                 }
716             }
717 
718             return *this;
719         }
720 
```

```
721         video::IImage* createHeightMap( ) const
722         {
723             dbPRINT( "CMatrix::createHeightMap()\n" );
724 
725             core::dimension2du img_size( Cols, Rows );
726             video::CImage* img = new video::CImage( video::ECF_A8R8G8B8, img_size);
727             if (!img)
728                 return 0;
729 
730             img->fill( 0xffffffff );
731 
732             const core::vector2df mm = getMinMax();
733             const ElementType height = mm.Y - mm.X;
734 
735             for (u32 y = 0; y < core::min_<u32>(Rows, img->getDimension().Height); y
++)
736             {
737                 for (u32 x = 0; x < core::min_<u32>(Cols, img->getDimension().Width
); x++)
738                 {
739                     ElementType value = Data[y][x];
740                     value -= mm.X;
741                     value /= height;
742                     value = core::clamp<ElementType>( value, 0.0f, 1.0f );
743                     video::SColorf color( value, value, value, 1.0f );
744                     img->setPixel( x, y, color.toSColor() );
745                 }
746             }
747 
748             return img;
749         }
750 
751         video::IImage* createImage( ) const
752         {
753             dbPRINT( "CMatrix::createImage()\n" );
754 
755             core::dimension2du img_size( Cols, Rows );
756             video::CImage* img = new video::CImage( video::ECF_A8R8G8B8, img_size);
757             if (!img)
758                 return 0;
759 
760             img->fill( 0xffffffff );
761 
762             const core::vector2df mm = getMinMax();
763             const ElementType height = mm.Y - mm.X;
764 
765             for (u32 y = 0; y < core::min_<u32>(Rows, img->getDimension().Height); y
++)
766             {
767                 for (u32 x = 0; x < core::min_<u32>(Cols, img->getDimension().Width
); x++)
768                 {
769                     ElementType value = Data[y][x];
770                     value -= mm.X;
771                     value /= height;
772                     value = core::clamp<ElementType>( value, 0.0f, 1.0f );
773                     video::SColorf color( value, value, value, 1.0f );
774                     img->setPixel( x, y, color.toSColor() );
775                 }
776             }
777 
778             return img;
779         }
780 
781         video::ITexture* createTexture( video::IVideoDriver* driver ) const
782         {
```

```
783             dbPRINT( "CMatrix::createTexture()\n" );
784
785             if (!driver)
786                 return 0;
787
788             video::IImage* img = createImage();
789
790             video::ITexture* tex = driver->addTexture( "createTexture", img, 0 );
791
792             return tex;
793         }
794
795
796         virtual ElementType det() const
797         {
798             return ElementType(0);
799         }
800
801 //      /// @brief Junk
802 //      /** Exchanges the pointers of 2 rows. */
803 //      bool shiftRow()
804 //      {
805 //  //      dbPRINT( "CMatrix::shiftRow()\n" );
806 //
807 //          u32 r = 1;
808 //          ElementType** b = new ElementType*[Rows];
809 //
810 //          if (!b)
811 //              return false;
812 //
813 //          u32 k = 0;
814 //          for (u32 i = r; i<Rows; i++)
815 //          {
816 //              b[i] = Data[k];
817 //              k++;
818 //          }
819 //
820 //          k = 0;
821 //          for (u32 i = Rows-r; i<Rows; i++)
822 //          {
823 //              b[k] = Data[i];
824 //              k++;
825 //          }
826 //
827 //          for (u32 i = 0; i<Rows; i++)
828 //          {
829 //              Data[i] = b[i];
830 //          }
831 //
832 //          delete [] b;
833 //
834 //          return true;
835 //      }
836
837     };
838
839     typedef CMatrix<f32> CMatrixf;
840
841 } // end namespace core
842 } // end namespace irr
843
844 #endif // __IRR_EXT_C_DYNAMIC_RECTANGLE_MATRIX_H__
```