```
#include <iostream>
#include "Errors.h"

#include "TYUV_file.h"

void print_stream_info(Yuv_file &YUV_file)
{
    std::cout << "resolution: "
            << YUV_file.get_resolution().x() << " "
            << YUV_file.get_resolution().y() << '\n';
    std::cout << "frame size: " << YUV_file.get_frame_size() << '\n';
    std::cout << "frames count: " << YUV_file.get_frames_count() << '\n';
}

int main(int argc, char *argv[]) try
{
    if(argc != 4)
        throw GeneralError("usage: yuvtool_cli res_x res_y input_file.yuv");            ;

    Yuv_file input_file(argv[3]);

    Vector<Unit::pixel> resolution;
    resolution.set_x(std::atoi(argv[1]));
    resolution.set_y(std::atoi(argv[2]));
    input_file.set_resolution(resolution);

    input_file.set_pixel_format(yuv_420p_8bit);

    print_stream_info(input_file);

    return 0;
}
catch(std::exception &e)
{
    std::cerr << "standard exception caught: " << e.what() << std::endl;
    return 1;
}
catch(...)
{
    std::cerr << "non-standard exception caught!" << std::endl;
    return 1;
}
```

```
#include <cassert>

#include <gui/drawer_gl.h>
#include <yuv/utils.h>
#include <yuv/picture_buffer.h>

const int tile_size = 64;

Drawer_gl::Drawer_gl()
{
}
//------------------------------------------------------------------------------
Drawer_gl::~Drawer_gl()
{
    reallocate_buffers( 0 );
}
//------------------------------------------------------------------------------
void Drawer_gl::draw( Yuv_file &yuv_file, int frame_number,
    Scroll_adapter &scroll_adapter )
{
//    const Gdk::Rectangle visible_area = scroll_adapter.get_visible_area();
//    const Coordinates tiles_start
//    {
//        visible_area.get_x()/tile_size,
//        visible_area.get_y()/tile_size
//    };
//    const Coordinates tiles_end
//    {
//        round_up( visible_area.get_x()+visible_area.get_width(), tile_size ) /
//            tile_size,
//        round_up( visible_area.get_y()+visible_area.get_height(), tile_size )
/
//            tile_size
//    };
//    const Coordinates tiles_counts
//    {
//        tiles_end.x - tiles_start.x,
//        tiles_end.y - tiles_start.y
//    };
//    reallocate_buffers( tiles_counts.x );

//    for( int tile_y=tiles_start.y; tile_y<tiles_end.y; tile_y++ )
//    {
//        const Coordinates buffer_start { tiles_start.x*tile_size,
//                tile_y*tile_size };
//        const Coordinates buffer_end { tiles_end.x*tile_size,
//                (tile_y+1)*tile_size };

//        const Picture_buffer coded_buffer =
//                yuv_file.extract_buffer(frame_number, buffer_start, buffer_end
);

//        for( int tile_x=tiles_start.x; tile_x<tiles_end.x; tile_x++ )
//        {
//            glBindBuffer( GL_PIXEL_UNPACK_BUFFER, m_buffers[tile_x] );
//            // glBufferData - to be sure storage is assigned and operations on
//            // old data are not going to stall us
//            // reserve storage only for RGB, not RGBA
//            glBufferData( GL_PIXEL_UNPACK_BUFFER,
//                Rgba_component_rgb_count * tile_size *
//                tile_size, 0, GL_STREAM_DRAW );
//            Byte *mapped_buffer = static_cast<Byte *>(
```



```
//                glMapBuffer( GL_PIXEL_UNPACK_BUFFER, GL_WRITE_ONLY ) );
//            if( !mapped_buffer )
//                throw std::runtime_error( "mapping of an OpenGL buffer failed"
//                    );
//            const Coordinates tile_start
//            {
//                tile_x*tile_size,
//                tile_y*tile_size
//            };
//            const Coordinates tile_end
//            {
//                (tile_x+1)*tile_size,
//                (tile_y+1)*tile_size
//            };
//            coded_buffer.fill_tile_rgb(tile_start, tile_end, mapped_buffer);
//            glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);

//            // TODO: draw square or triangle with m_buffer[tile_x] and
//            // m_texture[tile_x]
//            // glBindTexture, glTexSubImage2D, draw

//            glBindBuffer( GL_PIXEL_UNPACK_BUFFER, 0 );
//        }
//    }
//    reallocate_buffers( 0 );
}
//------------------------------------------------------------------------------
void Drawer_gl::reallocate_buffers( int buffers_count )
{
    assert( m_buffers.size() == m_textures.size() );
    const int old_size = m_buffers.size();
    if( old_size != buffers_count )
    {
        if( old_size < buffers_count )
        {
            m_buffers.resize( buffers_count );
            m_textures.resize( buffers_count );
            glGenBuffers( buffers_count-old_size, &m_buffers[old_size] );
            glGenTextures( buffers_count-old_size, &m_textures[old_size] );
        }
        else
        {
            glDeleteBuffers( old_size-buffers_count,
                &m_buffers[buffers_count] );
            glDeleteTextures( old_size-buffers_count,
                &m_textures[buffers_count] );
            m_buffers.resize( buffers_count );
            m_textures.resize( buffers_count );
        }
    }
}
```

```
#ifndef DRAWER_GL_H
#define DRAWER_GL_H

#include <yuv/TYUV_file.h>
#include <gui/scroll_adapter.h>
#include <GL/gl.h>

class Drawer_gl
{
public:
    Drawer_gl();
    ~Drawer_gl();
    void draw( Yuv_file &yuv_file, int frame_number,
        Scroll_adapter &scroll_adapter );

private:
    void reallocate_buffers( int buffers_count );

    //const int m_tile_size = 64;
    std::vector<GLuint> m_buffers;
    std::vector<GLuint> m_textures;
};

#endif // DRAWER_GL_H
```

```
#include <gtkmm/main.h>
#include "viewer_frame.h"

int main( int argc, char *argv[] )
{
    Gtk::Main kit( argc, argv );

    Viewer_frame viewer_frame;

    Gtk::Main::run( viewer_frame );

    return 0;
}
```

```cpp
#include "scroll_adapter.h"
#include <iostream>

Scroll_adapter::Scroll_adapter()
{
    get_hadjustment()->signal_changed().connect( sigc::mem_fun( *this,
        &Scroll_adapter::on_scroll ) );
    get_hadjustment()->signal_value_changed().connect( sigc::mem_fun( *this,
        &Scroll_adapter::on_scroll ) );
    get_vadjustment()->signal_changed().connect( sigc::mem_fun( *this,
        &Scroll_adapter::on_scroll ) );
    get_vadjustment()->signal_value_changed().connect( sigc::mem_fun( *this,
        &Scroll_adapter::on_scroll ) );

    Gtk::ScrolledWindow::add( m_fixed );
}
//-------------------------------------------------------------------------------
void Scroll_adapter::add( Gtk::Widget &widget )
{
    m_fixed.put( widget, 0, 0 );
}
//-------------------------------------------------------------------------------
void Scroll_adapter::remove()
{
    if(!m_fixed.get_children().empty())
        m_fixed.remove(*m_fixed.get_children().front());
    else
        std::cerr << "trying to remove child from scroll adapter, when none "
                "present\n";
}
//-------------------------------------------------------------------------------
void Scroll_adapter::remove( Gtk::Widget &widget )
{
    m_fixed.remove( widget );
}
//-------------------------------------------------------------------------------
void Scroll_adapter::set_internal_size( int width, int height )
{
    m_fixed.set_size_request( width, height );
}
//-------------------------------------------------------------------------------
void Scroll_adapter::get_internal_size( int &width, int &height ) const
{
    m_fixed.get_size_request( width, height );
}
//-------------------------------------------------------------------------------
Gdk::Rectangle Scroll_adapter::get_visible_area()
{
    int max_width, max_height;
    m_fixed.get_size_request( max_width, max_height );
    Gtk::Allocation result;
    result.set_x( get_hadjustment()->get_value() );
    result.set_y( get_vadjustment()->get_value() );
    result.set_width( std::min( max_width,
        static_cast<int>( get_hadjustment()->get_page_size() ) ) );
    result.set_height( std::min( max_height,
        static_cast<int>( get_vadjustment()->get_page_size() ) ) );
    return result;
}
//-------------------------------------------------------------------------------
void Scroll_adapter::on_scroll()
{
```



```cpp
    if( m_fixed.get_children().size() > 0 )
    {
        Gtk::Widget &adapted = *m_fixed.get_children().front();
        Gdk::Rectangle visible_area = get_visible_area();
        m_fixed.move( adapted, visible_area.get_x(), visible_area.get_y() );
        adapted.set_size_request( visible_area.get_width(),
            visible_area.get_height() );
    }
}
```

```
#ifndef SCROLL_ADAPTER_H
#define SCROLL_ADAPTER_H

#include <gtkmm.h>

class Scroll_adapter : public Gtk::ScrolledWindow {

public:
  Scroll_adapter();
  void add( Gtk::Widget &widget );
  void remove();
  void remove( Gtk::Widget &widget );
  void set_internal_size( int width, int height );
  void get_internal_size( int &width, int &height ) const;
  Gdk::Rectangle get_visible_area();
private:
  void on_scroll();

  Gtk::Fixed m_fixed;
};

#endif // SCROLL_ADAPTER_H
```

```
/* copied from https://github.com/LaurentGomila/SFML/wiki/Source%3A-GTK-SFMLWidg
et#wiki-sfmlwidgeth
 * public domain */

#include "SFMLWidget.h"
#include <iostream>

// Tested on Linux Mint 12.4 and Windows 7
#if defined(SFML_SYSTEM_WINDOWS)

#include <gdk/gdkwin32.h>
#define GET_WINDOW_HANDLE_FROM_GDK GDK_WINDOW_HANDLE

#elif defined(SFML_SYSTEM_LINUX) || defined(SFML_SYSTEM_FREEBSD)

#include <gdk/gdkx.h>
#define GET_WINDOW_HANDLE_FROM_GDK GDK_WINDOW_XID

#elif defined(SFML_SYSTEM_MACOS)

#error Note: You have to figure out an analogue way to access the handle of the
widget on a Mac-System

#else

#error Unsupported Operating System

#endif


SFMLWidget::SFMLWidget(sf::VideoMode mode, int size_request)
{
    if(size_request<=0)
        size_request = std::max<int>(1, std::min<int>(mode.width, mode.height) /
 2);

    set_size_request(size_request, size_request);

    set_has_window(false); // Makes this behave like an interal object rather th
en a parent window.
}

SFMLWidget::~SFMLWidget()
{
}

void SFMLWidget::on_size_allocate(Gtk::Allocation& allocation)
{
    //Do something with the space that we have actually been given:
    //(We will not be given heights or widths less than we have requested, thoug
h
    //we might get more)

    this->set_allocation(allocation);

    if(m_refGdkWindow)
    {
        m_refGdkWindow->move_resize(allocation.get_x(),
                                    allocation.get_y(),
                                    allocation.get_width(),
                                    allocation.get_height() );
        renderWindow.setSize(sf::Vector2u(allocation.get_width(),
```



```
                                          allocation.get_height()));
    }
}

void SFMLWidget::on_realize()
{
    Gtk::Widget::on_realize();

    if(!m_refGdkWindow)
    {
        //Create the GdkWindow:
        GdkWindowAttr attributes;
        memset(&attributes, 0, sizeof(attributes));

        Gtk::Allocation allocation = get_allocation();

        //Set initial position and size of the Gdk::Window:
        attributes.x = allocation.get_x();
        attributes.y = allocation.get_y();
        attributes.width = allocation.get_width();
        attributes.height = allocation.get_height();

        attributes.event_mask = get_events () | Gdk::EXPOSURE_MASK;
        attributes.window_type = GDK_WINDOW_CHILD;
        attributes.wclass = GDK_INPUT_OUTPUT;


        m_refGdkWindow = Gdk::Window::create(get_window(), &attributes,
                GDK_WA_X | GDK_WA_Y);
        set_has_window(true);
        set_window(m_refGdkWindow);

        // transparent background
#if GTK_VERSION_GE(3, 0)
        this->unset_background_color();
#else
        this->get_window()->set_back_pixmap(Glib::RefPtr<Gdk::Pixmap>());
#endif

        this->set_double_buffered(false);

        //make the widget receive expose events
        m_refGdkWindow->set_user_data(gobj());

        renderWindow.create(GET_WINDOW_HANDLE_FROM_GDK(m_refGdkWindow->gobj()));
    }
}

void SFMLWidget::on_unrealize()
{
  m_refGdkWindow.clear();

  //Call base class:
  Gtk::Widget::on_unrealize();
}

void SFMLWidget::display()
{
    if(m_refGdkWindow)
    {
        renderWindow.display();
    }
```

```
}

void SFMLWidget::invalidate()
{
    if(m_refGdkWindow)
    {
        m_refGdkWindow->invalidate(true);
    }
}
```

```
/* copied from https://github.com/LaurentGomila/SFML/wiki/Source%3A-GTK-SFMLWidg
et#wiki-sfmlwidgeth
 * public domain */

#ifndef SFMLWIDGET_H_INCLUDED
#define SFMLWIDGET_H_INCLUDED

#include <SFML/Graphics.hpp>
#include <gtkmm/widget.h>

class SFMLWidget : public Gtk::Widget
{
protected:
    sf::VideoMode m_vMode;

    virtual void on_size_allocate(Gtk::Allocation& allocation);
    virtual void on_realize();
    virtual void on_unrealize();

    Glib::RefPtr<Gdk::Window> m_refGdkWindow;
public:
    sf::RenderWindow renderWindow;

    SFMLWidget(sf::VideoMode mode, int size_request=-1);
    virtual ~SFMLWidget();

    void invalidate();
    void display();
};

#endif
```

```
#include "viewer_frame.h"
#include <iostream>
#include <cstdlib>
#include <SFML/OpenGL.hpp>

Viewer_frame::Viewer_frame() :
    m_box( Gtk::ORIENTATION_VERTICAL ),
    m_drawing_area( sf::VideoMode( 50, 50 ) )
{
    m_action_group = Gtk::ActionGroup::create();
    m_action_group->add(
        Gtk::Action::create( "menu_file", "_File" ) );
    m_action_group->add(
        Gtk::Action::create( "open_file", Gtk::Stock::OPEN ),
        sigc::mem_fun( *this, &Viewer_frame::on_action_file_open ) );
    m_action_group->add(
        Gtk::Action::create( "close_file", Gtk::Stock::CLOSE ),
        sigc::mem_fun( *this, &Viewer_frame::on_action_file_close ) );
    m_action_group->add(
        Gtk::Action::create( "quit", Gtk::Stock::QUIT ),
        sigc::mem_fun( *this, &Viewer_frame::on_action_file_quit ) );
    m_action_group->add(
        Gtk::Action::create( "show_size", Gtk::Stock::ABOUT,
            "Show size of the drowing area" ),
        sigc::mem_fun( *this, &Viewer_frame::on_action_show_size ) );

    m_ui_manager = Gtk::UIManager::create();
    m_ui_manager->insert_action_group( m_action_group );
    add_accel_group( m_ui_manager->get_accel_group() );
    Glib::ustring ui_info =
        R"(
   <ui>
     <menubar name='menu_bar'>
       <menu action='menu_file'>
         <menuitem action='open_file'/>
         <menuitem action='close_file'/>
         <separator/>
         <menuitem action='show_size'/>
         <separator/>
         <menuitem action='quit'/>
       </menu>
     </menubar>
     <toolbar name='tool_bar'>
       <toolitem action='open_file'/>
       <toolitem action='close_file'/>
       <toolitem action='show_size'/>
       <toolitem action='quit'/>
     </toolbar>
   </ui>
   )";
    m_ui_manager->add_ui_from_string( ui_info );

    set_default_size( 400, 250 );

    Gtk::Widget *menu_bar = m_ui_manager->get_widget("/menu_bar");
    m_box.pack_start( *menu_bar, Gtk::PACK_SHRINK );

    Gtk::Widget *tool_bar = m_ui_manager->get_widget("/tool_bar");
    m_box.pack_start( *tool_bar, Gtk::PACK_SHRINK );

    m_drawing_area.signal_configure_event().connect(
        sigc::mem_fun( *this, &Viewer_frame::on_action_configure_event ) );
```



```
    m_drawing_area.signal_size_allocate().connect(
        sigc::mem_fun( *this, &Viewer_frame::on_action_size_allocation ) );
    m_drawing_area.signal_draw().connect(
        sigc::mem_fun( *this, &Viewer_frame::on_action_draw_event ) );
    m_scroll_adapter.add( m_drawing_area );
    m_scroll_adapter.set_internal_size( 300, 300 );
    m_box.pack_start( m_scroll_adapter, Gtk::PACK_EXPAND_WIDGET );

    add(m_box);

    show_all();
}
//------------------------------------------------------------------------------
Viewer_frame::~Viewer_frame()
{ }
//------------------------------------------------------------------------------
void Viewer_frame::on_action_file_quit()
{
    hide();
}
//------------------------------------------------------------------------------
void Viewer_frame::on_action_show_size()
{
    Gtk::MessageDialog dialog( *this, "Size of the drawing area." );

    std::stringstream ss;
    ss << "The size of the drawing area is " <<
        m_drawing_area.get_width() << 'x' <<
        m_drawing_area.get_height() << '\n';
    int internal_width, internal_height;
    m_scroll_adapter.get_internal_size( internal_width, internal_height );
    ss << "The internal size of the scrolled area is " <<
        internal_width << 'x' <<
        internal_height << '\n';
    dialog.set_secondary_text( ss.str() );

    dialog.run();
}
//------------------------------------------------------------------------------
void Viewer_frame::on_action_file_open()
{
    Gtk::FileChooserDialog dialog( *this, "Choose YUV file.",
        Gtk::FILE_CHOOSER_ACTION_OPEN );

    dialog.add_button(Gtk::Stock::CANCEL, Gtk::RESPONSE_CANCEL);
    dialog.add_button("Select", Gtk::RESPONSE_OK);

    const int result = dialog.run();

    switch( result )
    {
    case Gtk::RESPONSE_OK:
        try
        {
            m_yuv_file.open( dialog.get_filename() );
        }
        catch( std::runtime_error & )
        {
            std::cerr << "failed to open file: " << dialog.get_filename();
        }
        break;
    case Gtk::RESPONSE_CANCEL:
```

```
        break;
    default:
        std::cerr << "unknown responce of file chooser dialog\n";
        break;
    }
}
//------------------------------------------------------------------------------
void Viewer_frame::on_action_file_close()
{
    if( m_yuv_file.is_open() )
    {
        m_yuv_file.close();
    }
    else
    {
        std::cerr << "tried to close file, while none is open\n";
    }
}
//------------------------------------------------------------------------------
bool Viewer_frame::on_action_configure_event( GdkEventConfigure* event )
{
    const int width = m_drawing_area.get_width();
    const int height = m_drawing_area.get_height();
    const int x0 = m_scroll_adapter.get_hadjustment()->get_value();
    const int y0 = m_scroll_adapter.get_vadjustment()->get_value();

    if( !m_drawing_area.renderWindow.setActive( true ) )
        return true;

    glViewport( 0, 0, width, height );

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glOrtho( x0, x0+width, y0+height, y0, -1, 1 );

    glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );
    glDisable( GL_DEPTH_TEST );

    if( !m_drawing_area.renderWindow.setActive( false ) )
        return true;

    return true;
}
//------------------------------------------------------------------------------
void Viewer_frame::on_action_size_allocation(
    Gtk::Allocation &allocation )
{
    on_action_configure_event( 0 );
}
//------------------------------------------------------------------------------
void Viewer_frame::draw_triangle()
{
    if( !m_drawing_area.renderWindow.setActive( true ) )
        return;

    glClearColor( 0.0, 0.0, 0.0, 0.0 );
    glClear( GL_COLOR_BUFFER_BIT );

    glColor3f( 1.0, 1.0, 1.0 );
    glBegin( GL_TRIANGLES );
    glVertex3f(  0.0,  0.0, 0.0 );
    glVertex3f( 20.0, 40.0, 0.0 );
```



```
    glVertex3f( 40.0,  0.0, 0.0 );
    glEnd();

    glFlush();
    m_drawing_area.display();

    if( !m_drawing_area.renderWindow.setActive( false ) )
        return;
}
//------------------------------------------------------------------------------
void Viewer_frame::draw_frame()
{
    if( !m_drawing_area.renderWindow.setActive( true ) )
        return;

    glClearColor( 0.0, 0.0, 0.0, 0.0 );
    glClear( GL_COLOR_BUFFER_BIT );
    glEnable( GL_TEXTURE_2D );
    glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL );

    m_drawer_gl.draw( m_yuv_file, 0, m_scroll_adapter );

    glFlush();
    m_drawing_area.display();

    if( !m_drawing_area.renderWindow.setActive( false ) )
        return;
}
//------------------------------------------------------------------------------
bool Viewer_frame::on_action_draw_event( const::Cairo::RefPtr<Cairo::Context>
  &cr )
{
    draw_triangle();
    //draw_frame();
    return true;
}
```

```
#ifndef VIEWER_FRAME_H
#define VIEWER_FRAME_H

#include <gui/SFMLWidget.h>
#include <gtkmm.h>
#include <gtkmm/socket.h>
#include <yuv/TYUV_file.h>
#include <gui/scroll_adapter.h>
#include <gui/drawer_gl.h>

class Viewer_frame : public Gtk::Window
{
public:
    Viewer_frame();
    virtual ~Viewer_frame();

private:
    void on_action_file_quit();
    void on_action_show_size();
    void on_action_file_open();
    void on_action_file_close();
    bool on_action_configure_event( GdkEventConfigure *event );
    void on_action_size_allocation( Gtk::Allocation &allocation );
    bool on_action_draw_event( const ::Cairo::RefPtr< ::Cairo::Context>& cr );
    void draw_triangle();
    void draw_frame();

    Glib::RefPtr<Gtk::ActionGroup> m_action_group;
    Glib::RefPtr<Gtk::UIManager> m_ui_manager;
    Gtk::Box m_box;
    SFMLWidget m_drawing_area;
    Scroll_adapter m_scroll_adapter;
    Drawer_gl m_drawer_gl;

    Yuv_file m_yuv_file;
};

#endif // VIEWER_FRAME_H
```

```
#ifndef YUV_TOOL_APPLICATION_H
#define YUV_TOOL_APPLICATION_H

#endif // YUV_TOOL_APPLICATION_H
```

```
#ifndef BIT_POSITION_H
#define BIT_POSITION_H

#include <yuv/utils.h>
#include <ostream>

class Bit_position
{
private:
    typedef int Position_type;
    Position_type m_position;
public:
    Bit_position(Position_type position = 0) :
        m_position(position)
    { }
//------------------------------------------------------------------------------
    Bit_position(const int bytes, const int bits) :
        m_position(bytes*bits_in_byte + bits)
    { }
//------------------------------------------------------------------------------
    Position_type get_position() const
    {
        return m_position;
    }
//------------------------------------------------------------------------------
    void set_position(const Position_type position)
    {
        m_position = position;
    }
//------------------------------------------------------------------------------
    Position_type get_bytes() const
    {
        return m_position / bits_in_byte;
    }
//------------------------------------------------------------------------------
    int get_bits() const
    {
        return m_position % bits_in_byte;
    }
//------------------------------------------------------------------------------
    void set_position(const Position_type byte, const int bit)
    {
        set_position(byte * bits_in_byte + bit);
    }
};
//------------------------------------------------------------------------------
inline Bit_position &operator+=(Bit_position &a, const Bit_position &b)
{
    a.set_position(a.get_position() + b.get_position());
    return a;
}
//------------------------------------------------------------------------------
inline Bit_position &operator-=(Bit_position &a, const Bit_position &b)
{
    a.set_position(a.get_position() - b.get_position());
    return a;
}
//------------------------------------------------------------------------------
inline Bit_position operator+(const Bit_position &a, const Bit_position &b)
{
    Bit_position result = a;
    result += b;
```



```
    return result;
}
//------------------------------------------------------------------------------
inline Bit_position operator-(const Bit_position &a, const Bit_position &b)
{
    Bit_position result = a;
    result -= b;
    return result;
}
//------------------------------------------------------------------------------
inline Bit_position &operator*=(Bit_position &a, const int factor)
{
    a.set_position(a.get_position()*factor);
    return a;
}
//------------------------------------------------------------------------------
inline Bit_position operator*(const Bit_position &a, const int factor)
{
    Bit_position result = a;
    result *= factor;
    return result;
}
//------------------------------------------------------------------------------
inline Bit_position operator*(const int factor, const Bit_position &a)
{
    return a*factor;
}
//------------------------------------------------------------------------------
inline int operator/(const Bit_position &a, const Bit_position &b)
{
    return a.get_position()/b.get_position();
}
//------------------------------------------------------------------------------
inline bool operator<(const Bit_position &a, const Bit_position &b)
{
    return a.get_position() < b.get_position();
}
//------------------------------------------------------------------------------
inline bool operator>(const Bit_position &a, const Bit_position &b)
{
    return a.get_position() > b.get_position();
}
//------------------------------------------------------------------------------
inline std::ostream &operator<<(std::ostream &stream, const Bit_position &a)
{
    stream << a.get_bytes() << "bytes, " << a.get_bits() << "bits";
    return stream;
}
//------------------------------------------------------------------------------
inline Bit_position &operator++(Bit_position &a)
{
    a.set_position(a.get_position() + 1);
    return a;
}
//------------------------------------------------------------------------------
inline Bit_position operator++(Bit_position &a, const int)
{
    Bit_position tmp(a);
    ++a;
    return tmp;
}
//------------------------------------------------------------------------------
```

```
inline Bit_position &operator--(Bit_position &a)
{
    a.set_position(a.get_position() - 1);
    return a;
}
//------------------------------------------------------------------------------
inline Bit_position operator--(Bit_position &a, const int)
{
    Bit_position tmp(a);
    --a;
    return tmp;
}

#endif // BIT_POSITION_H
```

```
#ifndef COORDINATES_H
#define COORDINATES_H

class Coordinate_pair
{
public:
    typedef int Scalar;
private:
    Scalar m_x;
    Scalar m_y;
public:
    Coordinate_pair(const Scalar x = 0, const Scalar y = 0)
    {
        set(x, y);
    }
    void set(const Scalar x = 0, const Scalar y = 0)
    {
        set_x(x);
        set_y(y);
    }
    void set_x(const Scalar x)
    {
        m_x = x;
    }
    void set_y(const Scalar y)
    {
        m_y = y;
    }
    Scalar x() const
    {
        return m_x;
    }
    Scalar y() const
    {
        return m_y;
    }
};
//--------------------------------------------------------------------------------
inline Coordinate_pair &operator+=(
        Coordinate_pair &a,
        const Coordinate_pair &b)
{
    a.set(
            a.x() + b.x(),
            a.y() + b.y());
    return a;
}
//--------------------------------------------------------------------------------
inline Coordinate_pair operator+(
        Coordinate_pair a,
        const Coordinate_pair &b)
{
    return a += b;
}
//--------------------------------------------------------------------------------
inline Coordinate_pair &operator-=(
        Coordinate_pair &a,
        const Coordinate_pair &b)
{
    a.set(
            a.x() - b.x(),
            a.y() - b.y());
```



```
    return a;
}
//--------------------------------------------------------------------------------
inline Coordinate_pair operator-(
        Coordinate_pair a,
        const Coordinate_pair &b)
{
    return a -= b;
}
//--------------------------------------------------------------------------------
inline bool operator==(
        const Coordinate_pair &a,
        const Coordinate_pair &b)
{
    return
            a.x() == b.x()
            && a.y() == b.y();
}
//--------------------------------------------------------------------------------
inline bool operator!=(
        const Coordinate_pair &a,
        const Coordinate_pair &b )
{
    return !(a == b);
}
//--------------------------------------------------------------------------------
enum class Unit
{
    pixel,
    macropixel
};
//--------------------------------------------------------------------------------
enum class Reference_point
{
    macropixel,
    picture
};
//--------------------------------------------------------------------------------
template<Unit unit>
class Vector : private Coordinate_pair
{
public:
    using Coordinate_pair::Scalar;
    Vector(const Scalar x = 0, const Scalar y = 0) :
        Coordinate_pair(x, y)
    { }
    using Coordinate_pair::set;
    using Coordinate_pair::set_x;
    using Coordinate_pair::set_y;
    using Coordinate_pair::x;
    using Coordinate_pair::y;

    Vector &operator+=(const Vector &b)
    {
        *static_cast<Coordinate_pair *>(this) += b;
        return *this;
    }
    Vector &operator-=(const Vector &b)
    {
        *static_cast<Coordinate_pair *>(this) -= b;
        return *this;
    }
```

```
    bool operator==(const Vector &b) const
    {
        return *static_cast<const Coordinate_pair *>(this) == b;
    }
    bool operator!=(const Vector &b) const
    {
        return *static_cast<const Coordinate_pair *>(this) != b;
    }
    template<Unit coordinate_unit, Reference_point reference_point>
    friend class Coordinates;
private:
    Vector(const Coordinate_pair &coordinate_pair) :
        Coordinate_pair(coordinate_pair)
    { }
};
//---------------------------------------------------------------------------------
template<Unit unit>
Vector<unit> operator+(Vector<unit> a, const Vector<unit> &b)
{
    return a += b;
}
//---------------------------------------------------------------------------------
template<Unit unit>
Vector<unit> operator-(Vector<unit> a, const Vector<unit> &b)
{
    return a -= b;
}
//---------------------------------------------------------------------------------
template<Unit unit, Reference_point reference_point>
class Coordinates : private Coordinate_pair
{
public:
    using Coordinate_pair::Scalar;
    Coordinates(const Scalar x = 0, const Scalar y = 0) :
        Coordinate_pair(x, y)
    { }
    using Coordinate_pair::set;
    using Coordinate_pair::set_x;
    using Coordinate_pair::set_y;
    using Coordinate_pair::x;
    using Coordinate_pair::y;

    Coordinates &operator+=(const Vector<unit> &v)
    {
        *static_cast<Coordinate_pair *>(this) += v;
        return *this;
    }
    Coordinates &operator-=(const Vector<unit> &v)
    {
        *static_cast<Coordinate_pair *>(this) -= v;
        return *this;
    }
    Vector<unit> operator-(const Coordinates &b) const
    {
        return Vector<unit>(*static_cast<const Coordinate_pair *>(this) - b);
    }
    bool operator==(const Coordinates &b) const
    {
        return *static_cast<const Coordinate_pair *>(this) == b;
    }
    bool operator!=(const Coordinates &b) const
    {
```



```
        return *static_cast<const Coordinate_pair *>(this) != b;
    }
};
//---------------------------------------------------------------------------------
template<Unit unit, Reference_point reference_point>
Coordinates<unit, reference_point> operator+(
        Coordinates<unit, reference_point> a,
        const Vector<unit> &v)
{
    return a += v;
}
//---------------------------------------------------------------------------------
template<Unit unit, Reference_point reference_point>
Coordinates<unit, reference_point> operator-(
        Coordinates<unit, reference_point> a,
        const Vector<unit> &v)
{
    return a -= v;
}
//---------------------------------------------------------------------------------
inline Vector<Unit::pixel> cast_to_pixels(
        const Vector<Unit::macropixel> &a,
        const Vector<Unit::pixel> &macropixel_size)
{
    return Vector<Unit::pixel>(
            a.x() * macropixel_size.x(),
            a.y() * macropixel_size.y());
}
//---------------------------------------------------------------------------------
inline Vector<Unit::macropixel> cast_to_macropixels(
        const Vector<Unit::pixel> &a,
        const Vector<Unit::pixel> &macropixel_size,
        Vector<Unit::pixel> &remainder)
{
    Vector<Unit::macropixel> result(
            a.x() / macropixel_size.x(),
            a.y() / macropixel_size.y());
    remainder = a - cast_to_pixels(result, macropixel_size);
    return result;
}
//---------------------------------------------------------------------------------
template<Reference_point reference_point>
Coordinates<Unit::pixel, reference_point> cast_to_pixels(
        const Coordinates<Unit::macropixel, reference_point> &a,
        const Vector<Unit::pixel> &macropixel_size,
        const Coordinates<Unit::pixel, Reference_point::macropixel> &remainder =
            Coordinates<Unit::pixel, Reference_point::macropixel>(0, 0))
{
    return
            Coordinates<Unit::pixel, reference_point>(
                a.x() * macropixel_size.x(),
                a.y() * macropixel_size.y())
            + (
                remainder
                - Coordinates<Unit::pixel, Reference_point::macropixel>(0, 0));
}
//---------------------------------------------------------------------------------
template<Reference_point reference_point>
Coordinates<Unit::macropixel, reference_point> cast_to_macropixels(
        const Coordinates<Unit::pixel, reference_point> &a,
        const Vector<Unit::pixel> &macropixel_size,
        Coordinates<Unit::pixel, Reference_point::macropixel> &remainder)
```

```
{
    Coordinates<Unit::macropixel, reference_point> result(
            a.x() / macropixel_size.x(),
            a.y() / macropixel_size.y());
    // TODO: write and use reference cast
    remainder =
            Coordinates<Unit::pixel, Reference_point::macropixel>(0, 0)
            + (a - cast_to_pixels(result, macropixel_size));
    return result;
}
//-------------------------------------------------------------------------------
template<Unit Tunit, Reference_point Tpoint>
class Coordinate_range
{
public:
    static constexpr Unit unit = Tunit;
    static constexpr Reference_point reference_point = Tpoint;

private:
    Coordinates<unit, reference_point> m_start;
    Vector<unit> m_size;

public:
    class Iterator
    {
    private:
        const Coordinate_range &m_range;
        Coordinates<unit, reference_point> m_coordinates;

    private:
        Iterator(
                const Coordinate_range &range,
                const Coordinates<unit, reference_point> &coordinates) :
            m_range(range),
            m_coordinates(coordinates)
        { }
    public:
        Coordinates<unit, reference_point> operator*() const
        {
            return m_coordinates;
        }
        Iterator operator++()
        {
            Iterator result = *this;
            m_coordinates += Vector<unit>(1, 0);
            if(m_coordinates.x() >= m_range.m_start.x() + m_range.m_size.x())
                m_coordinates += Vector<unit>(-m_range.m_size.x(), 1);
            return result;
        }
        bool operator!=(const Iterator &other) const
        {
            return m_coordinates != other.m_coordinates;
        }
        friend class Coordinate_range;
    };

    Coordinate_range(
            const Coordinates<unit, reference_point> &start,
            const Vector<unit> &size) :
        m_start(start),
        m_size(size)
    { }
```



```
    Iterator begin() const
    {
        return Iterator(*this, m_start);
    }
    Iterator end() const
    {
        return Iterator(*this, m_start + Vector<unit>(0, m_size.y()));
    }
};

#endif // COORDINATES_H
```

```
#include "Errors.h"


GeneralError::GeneralError(const std::string &description) :
        m_description(description)
{ }

const char *GeneralError::what() const throw()
{
    return ("general error: " + m_description).c_str();
}




//TIoError::TIoError( const std::string &file_name ) :
//  m_file_name( file_name )
//{ ; }
//
//const char *TIoError::what() {
//  return ""
//}
```

```
#ifndef ERRORS_H
#define ERRORS_H

#include <exception>
#include <stdexcept>
#include <string>

class GeneralError : public std::exception
{
public:
    GeneralError(const std::string &file_name);
    const char* what() const throw();
private:
    std::string m_description;
};

template<typename Exception = std::runtime_error, typename Argument>
void my_assert(const bool test, const Argument &argument)
{
    if(!test)
        throw Exception(argument);
}

inline void check_range(int begin, int value, int end)
{
    my_assert<std::out_of_range>(
            begin <= value && value < end,
            "YUVtool variable");
}

#endif // ERRORS_H
```

```
#include <algorithm>
#include <stdexcept>

#include "picture_buffer.h"
#include "Errors.h"

namespace
{
int get_bits(const Byte *data, const Bit_position start, const Bit_position end)
{
    // TODO: optimize!!!
    int result = 0;
    int mask = 1;
    for(Bit_position i = start; i < end; ++i)
    {
        const int bit = (data[i.get_bytes()] >> i.get_bits()) & 1;
        result |= bit ? mask : 0;
        mask <<= 1;
    }
    return result;
}
//-----------------------------------------------------------------------------
void set_bits(Byte *data, const Bit_position start, const Bit_position end,
        const int value)
{
    // TODO: optimize!!!
    int mask = 1;
    for(Bit_position i = start; i < end; ++i)
    {
        const int bit = value & mask ? 1 : 0;
        data[i.get_bytes()] |= bit << i.get_bits();
        mask <<= 1;
    }
}
}
//-----------------------------------------------------------------------------
Picture_buffer::Picture_buffer(
        const Vector<Unit::pixel> &resolution,
        const Pixel_format &pixel_format)
{
    allocate(resolution, pixel_format);
}
//-----------------------------------------------------------------------------
Picture_buffer::~Picture_buffer()
{
}
//-----------------------------------------------------------------------------
void Picture_buffer::allocate(
        const Vector<Unit::pixel> &resolution,
        const Pixel_format &pixel_format)
{
    m_parameters.recalculate(pixel_format, resolution);
    m_data.resize(m_parameters.get_buffer_size().get_bytes());
}
//-----------------------------------------------------------------------------
const Pixel_format &Picture_buffer::get_pixel_format() const
{
    return m_parameters.get_pixel_format();
}
//-----------------------------------------------------------------------------
const Vector<Unit::pixel> &Picture_buffer::get_resolution() const
{
```



```
    return m_parameters.get_resolution();
}
//-----------------------------------------------------------------------------
const Precalculated_buffer_parameters &Picture_buffer::get_parameters() const
{
    return m_parameters;
}
//-----------------------------------------------------------------------------
std::vector< Byte > &Picture_buffer::get_data()
{
    return m_data;
}
//-----------------------------------------------------------------------------
const std::vector< Byte > &Picture_buffer::get_data() const
{
    return m_data;
}
//-----------------------------------------------------------------------------
//void Picture_buffer::fill_tile_rgb(Coordinates tile_start,
//    Coordinates tile_end, Byte *data) const
//{
//    // data is assumed to be in form:
//    // format: GL_RGB
//    // type: GL_BYTE
//    // as given to glTexImage2D

//    const int output_bytes_per_pixel = 3;

//    const Coordinates macropixel_size = m_parameters.get_macropixel_size();
//    //const int bytes_per_pixel_component = 1;
//    Coordinates macropixel_coordinates;

//    for( macropixel_coordinates.y=tile_start.y;
//         macropixel_coordinates.y<tile_end.y;
//         macropixel_coordinates.y+=macropixel_size.y )
//    {
//        for( macropixel_coordinates.x=tile_start.x;
//             macropixel_coordinates.x<tile_end.x;
//             macropixel_coordinates.x+=macropixel_size.x )
//        {
//            const int tile_width = tile_end.x - tile_start.x;
//            const int pixel_number_in_tile =
//                tile_width * macropixel_coordinates.y +
//                macropixel_coordinates.x;
//            draw_macropixel( macropixel_coordinates,
//                data + output_bytes_per_pixel * pixel_number_in_tile );
//        }
//    }
//}
//-----------------------------------------------------------------------------
//void Picture_buffer::draw_macropixel( Coordinates coordinates, Byte *data )
//    const
//{
//    const Coordinates size = m_parameters.get_macropixel_size();
//    Coordinates pixel_offset;
//    for( pixel_offset.y=0;
//        pixel_offset.y<size.y;
//        pixel_offset.y++ )
//    {
//        for( pixel_offset.x=0;
//            pixel_offset.x<size.x;
//            pixel_offset.x++ )
```

```
//        {
//            double rgb_values[Rgba_component_rgb_count] = { 0 };
//            const Coded_pixel &coded_pixel =
//                get_pixel_format().m_macropixel_coding.m_pixels[
//                    pixel_offset.y*size.x+pixel_offset.x];
//            draw_pixel( coordinates+pixel_offset, coded_pixel, rgb_values );
//        }
//    }
//}
//--------------------------------------------------------------------------------
int Picture_buffer::get_entry(
        const Coordinates<Unit::pixel, Reference_point::picture> &coordinates,
        const int component_index) const
{
    Bit_position start =
            get_parameters().get_entry_offset(coordinates, component_index);
    Bit_position width =
            get_parameters().get_bits_per_entry(coordinates, component_index);
    return get_bits(get_data().data(), start, start+width);
}
//--------------------------------------------------------------------------------
void Picture_buffer::set_entry(
        const Coordinates<Unit::pixel, Reference_point::picture> &coordinates,
        const int component_index,
        const int value)
{
    Bit_position start =
            get_parameters().get_entry_offset(coordinates, component_index);
    Bit_position width =
            get_parameters().get_bits_per_entry(coordinates, component_index);
    return set_bits(get_data().data(), start, start+width, value);
}
//--------------------------------------------------------------------------------
void Picture_buffer::convert_color_space(
        const std::vector<Component> &components)
{
    my_assert(
            m_parameters.is_expanded(),
            "color space conversion not expanded pixel formats is not "
            "supported");
    if(components == m_parameters.get_pixel_format().m_components)
        return;

//    const int planes_count =  m_parameters.get_planes_count();
//    const Vector<Unit::pixel> &resolution = m_parameters.get_resolution();

    for(const auto &xy : m_parameters.get_pixel_range())
    {
        xy.x();
    }

    throw std::runtime_error("TODO");
}
//--------------------------------------------------------------------------------
//void Picture_buffer::draw_pixel( Coordinates coordinates,
//    const Coded_pixel &coded_pixel,
//    double (&result)[Rgba_component_rgb_count] ) const
//{
//    const Coordinates macropixel_size = m_parameters.get_macropixel_size();
//    const Coordinates coordinates_in_macropixel_units =
//            coordinates / macropixel_size;
//    const Coordinates picture_size_in_macropixel_units =
```



```
//              m_parameters.get_size_in_macropixels();

//    const int components_count = m_parameters.get_components_count();
//    for(int component_index = 0;
//        component_index < components_count;
//        component_index++)
//    {
//        const Component_coding &component_coding =
//            coded_pixel.m_components[component_index];
//        const Component &component =
//            m_pixel_format.m_components[component_index];
//        for( Rgba_component rgb_component = Rgba_component_red;
//            rgb_component != Rgba_component_rgb_count;
//            rgb_component = static_cast<Rgba_component>( rgb_component+1 ) )
//        {
//            const int plane_index = component_coding.m_plane_index;
//            const int entry_index = component_coding.m_entry_index;
//            const Plane &plane = m_pixel_format.m_planes[plane_index];
//            const Entry &entry = plane.m_entries[entry_index];

//            // TODO: move this fragment to separate function
//            // maybe cache the result on allocation
//            const int entry_rows_per_row_of_macropixels =
//                plane.m_entries.size() / plane.m_entries_per_macropixel_width;
//            const int entry_row = coordinates_in_macropixel_units.y *
//                entry_rows_per_row_of_macropixels +
//                entry_index / plane.m_entries_per_macropixel_width;
//            const int bits_per_row_of_entries = std::accumulate(
//                plane.m_entries.begin(),
//                plane.m_entries.begin()+plane.m_entries_per_macropixel_width,
//                0,
//                []( int sum, const Entry &entry )
//                {
//                    return sum+entry.m_bit_width;
//                } );
//            const int offset_from_left_edge_of_macropixel = std::accumulate(
//                plane.m_entries.begin(),
//                plane.m_entries.begin() +
//                    entry_index%plane.m_entries_per_macropixel_width,
//                0,
//                []( int sum, const Entry &entry )
//                {
//                    return sum+entry.m_bit_width;
//                } );
//            const int plane_offset_bits = m_plane_offsets[plane_index];
//            const int start_bit =
//                plane_offset_bits +
//                entry_row * bits_per_row_of_entries *
//                    picture_size_in_macropixel_units.x +
//                bits_per_row_of_entries * coordinates_in_macropixel_units.x +
//                offset_from_left_edge_of_macropixel;
//            const int end_bit = start_bit + entry.m_bit_width;
//            const int source_value = get_bits( m_data.data(),
//                start_bit, end_bit );
//            result[rgb_component] += component.m_coeff[rgb_component] *
//                source_value;
//        }
//    }
//}
//--------------------------------------------------------------------------------
Picture_buffer convert(
        const Picture_buffer &source,
```

```
        const Pixel_format &pixel_format)
{
    Picture_buffer expanded = expand_sampling(source);
    expanded.convert_color_space(pixel_format.m_components);
    throw std::runtime_error("TODO");
    return expanded;
}
//-----------------------------------------------------------------------------
Picture_buffer expand_sampling(const Picture_buffer &source)
{
    const Precalculated_buffer_parameters &source_parameters =
            source.get_parameters();
    const Vector<Unit::pixel> macropixel_size =
            source_parameters.get_macropixel_size();
    const Pixel_format expanded_format =
            get_expanded_pixel_format(source_parameters.get_pixel_format());
    Picture_buffer expanded(source.get_resolution(), expanded_format);
    const Precalculated_buffer_parameters &expanded_parameters =
            expanded.get_parameters();

    const Vector<Unit::macropixel> size_in_macropixels =
            source_parameters.get_size_in_macropixels();

    my_assert(
            cast_to_pixels(size_in_macropixels, macropixel_size)
            != source.get_resolution(),
            "not supported yet");

    for(int iy = 0; iy < size_in_macropixels.y(); iy++)
    {
        for(int jy = 0; jy < macropixel_size.y(); jy++)
        {
            for(int ix = 0; ix < size_in_macropixels.x(); ix++)
            {
                for(int jx = 0; jx < macropixel_size.x(); jx++)
                {
                    const Coordinates<Unit::pixel, Reference_point::picture>
                            coordinates(
                                ix * macropixel_size.x() + jx,
                                iy * macropixel_size.y() + jy);
                    const int components_count =
                            source_parameters.get_components_count();
                    for(
                            int component_index = 0;
                            component_index < components_count;
                            component_index++)
                    {
                        const Bit_position input_bitdepth =
                                source_parameters.get_bits_per_entry(
                                    Coordinates<Unit::pixel,
                                        Reference_point::macropixel>(jx, jy),
                                    component_index);
                        const Bit_position output_bitdepth =
                                expanded_parameters.get_bits_per_entry(
                                    Coordinates<Unit::pixel,
                                        Reference_point::macropixel>(0, 0),
                                    component_index);
                        const int input_value =
                                source.get_entry(coordinates, component_index);
                        const int output_value =
                                input_value
                                << (
```



```
                                    output_bitdepth.get_position()
                                    - input_bitdepth.get_position());
                        expanded.set_entry(
                                coordinates,
                                component_index,
                                output_value);
                    }
                }
            }
        }
    }
    return expanded;
}
```

```cpp
#ifndef PICTURE_BUFFER_H
#define PICTURE_BUFFER_H

#include <yuv/utils.h>
#include <yuv/TComponents.h>

class Picture_buffer
{
public:
    Picture_buffer()
    { }
    Picture_buffer(
            const Vector<Unit::pixel> &resolution,
            const Pixel_format &pixel_format);
    ~Picture_buffer();
    void allocate(
            const Vector<Unit::pixel> &resolution,
            const Pixel_format &pixel_format);
    const Pixel_format &get_pixel_format() const;
    const Vector<Unit::pixel> &get_resolution() const;
    const Precalculated_buffer_parameters &get_parameters() const;
    std::vector< Byte > &get_data();
    const std::vector< Byte > &get_data() const;
//    void fill_tile_rgb(Coordinates tile_start, Coordinates tile_end,
//        Byte *data ) const;
    int get_entry(
            const Coordinates<Unit::pixel, Reference_point::picture>
                &coordinates,
            int component_index) const;
    void set_entry(
            const Coordinates<Unit::pixel, Reference_point::picture>
                &coordinates,
            int component_index,
            int value);
    void convert_color_space(const std::vector<Component> &components);

private:
//    void draw_macropixel( Coordinates coordinates, Byte *data ) const;
//    void draw_pixel( Coordinates coordinates, const Coded_pixel &coded_pixel,
//        double (&result)[Rgba_component_rgb_count] ) const;

    Precalculated_buffer_parameters m_parameters;
    std::vector< Byte > m_data;
};

Picture_buffer convert(
        const Picture_buffer &source,
        const Pixel_format &pixel_format);
Picture_buffer expand_sampling(const Picture_buffer &source);

#endif // PICTURE_BUFFER_H
```

```
#include <yuv/TComponents.h>

#include <stdexcept>
#include <algorithm>

//-----------------------------------------------------------------------------
Pixel_format get_expanded_pixel_format(
        const std::vector<Component> &components,
        const std::vector<Entry> &entries)
{
    const int components_count = components.size();
    Pixel_format result;
    result.m_components = components;
    result.m_macropixel_coding.m_size = {1, 1};
    result.m_macropixel_coding.m_pixels.resize(1);
    Coded_pixel &target_pixel =
            result.m_macropixel_coding.m_pixels.front();
    target_pixel.m_components.resize(components_count);
    result.m_planes.resize(components_count);
    for(int component_index = 0; component_index < components_count;
            component_index++)
    {
        target_pixel.m_components[component_index] = {component_index, 0, 0};
        result.m_planes[component_index] = {{{{entries[component_index]}}}};
    }
    return result;
}
//-----------------------------------------------------------------------------
Pixel_format get_expanded_pixel_format(const Pixel_format &input)
{
    const int components_count = input.m_components.size();

    std::vector<Entry> entries(components_count);
    const std::vector<Coded_pixel> &coded_pixels =
            input.m_macropixel_coding.m_pixels;
    for(int plane_index = 0; plane_index < components_count; plane_index++)
    {
        const Bit_position max_depth =
                std::accumulate(
                    coded_pixels.begin(),
                    coded_pixels.end(),
                    Bit_position(0),
                    [plane_index, input](
                            const Bit_position max,
                            const Coded_pixel &coded_pixel)
                    {
                        const Component_coding &input_component_coding =
                                coded_pixel.m_components[plane_index];
                        const Entry &input_entry =
                                input.m_planes[
                                input_component_coding.m_plane_index].m_rows[
                                input_component_coding.m_row_index].m_entries[
                                input_component_coding.m_entry_index];
                        return std::max(max, input_entry.m_width);
                    });
        entries[plane_index].m_width.set_position(round_up(
                max_depth.get_position(), bits_in_byte));
    }

    return get_expanded_pixel_format(input.m_components, entries);
}
//-----------------------------------------------------------------------------
```



```
Precalculated_pixel_format::Precalculated_pixel_format()
{
    clear();
}
//-----------------------------------------------------------------------------
void Precalculated_pixel_format::clear()
{
    m_bits_per_macropixel = 0;
    m_planes.clear();
    m_is_expanded = false;
}
//-----------------------------------------------------------------------------
void Precalculated_pixel_format::recalculate(const Pixel_format &pixel_format)
{
    const int planes_count = pixel_format.m_planes.size();

    m_pixel_format = pixel_format;

    m_planes.resize(planes_count);
    for(int plane_index = 0; plane_index < planes_count; plane_index++)
    {
        Plane_parameters &plane_parameters = m_planes[plane_index];
        const Plane &plane = pixel_format.m_planes[plane_index];

        const int rows_count = pixel_format.m_planes[plane_index].m_rows.size();
        plane_parameters.m_rows.resize(rows_count);
        for(int row_index = 0; row_index < rows_count; row_index++)
        {
            Entry_row_paramters &row_parameters =
                    plane_parameters.m_rows[row_index];
            const Entry_row &row = plane.m_rows[row_index];

            const int entry_count = row.m_entries.size();
            row_parameters.m_entries.resize(entry_count);
            Bit_position offset = 0;
            for(int entry_index = 0; entry_index < entry_count; entry_index++)
            {
                Entry_parameters &entry = row_parameters.m_entries[entry_index];
                entry.m_offset = offset;
                entry.m_sampling_point = {-1, -1};
                offset += row.m_entries[entry_index].m_width;
            }
            row_parameters.m_bits_per_macropixel = offset;
        }
        plane_parameters.m_bits_per_macropixel =
                std::accumulate(
                    plane_parameters.m_rows.begin(),
                    plane_parameters.m_rows.end(),
                    Bit_position(0),
                    [](const Bit_position sum, const Entry_row_paramters &row)
                    {
                        return sum + row.m_bits_per_macropixel;
                    });
    }
    m_bits_per_macropixel =
            std::accumulate(
                m_planes.begin(),
                m_planes.end(),
                Bit_position(0),
                [](const Bit_position sum, const Plane_parameters &plane)
                {
                    return sum + plane.m_bits_per_macropixel;
```

```cpp
                });
    for(int iy = 0; iy < get_macropixel_size().y(); iy++)
    {
        for(int ix = 0; ix < get_macropixel_size().x(); ix++)
        {
            const Coordinates<Unit::pixel, Reference_point::macropixel>
                    coordinates(ix, iy);
            const int pixel_index = iy * get_macropixel_size().y() + ix;
            for(
                    int component_index = 0;
                    component_index < get_components_count();
                    component_index++)
            {
                const Component_coding &component_coding =
                        pixel_format.m_macropixel_coding.m_pixels[
                        pixel_index].m_components[
                        component_index];
                Entry_parameters &entry_parameters =
                        m_planes[
                        component_coding.m_plane_index].m_rows[
                        component_coding.m_row_index].m_entries[
                        component_coding.m_entry_index];
                if(
                        entry_parameters.m_sampling_point
                        != Coordinates<Unit::pixel,
                            Reference_point::macropixel>(-1, -1))
                {
                    entry_parameters.m_sampling_point = coordinates;
                }
            }
        }
    }

    // m_is_expanded
    if(planes_count == get_components_count())
    {
        for(int plane_index = 0; plane_index < planes_count; plane_index++)
        {
            for(const Coded_pixel &pixel : get_pixel_format(
                    ).m_macropixel_coding.m_pixels)
            {
                const Component_coding &component =
                        pixel.m_components[plane_index];
                if(
                        component.m_plane_index != plane_index
                        || component.m_row_index != 0
                        || component.m_entry_index != 0)
                {
                    m_is_expanded = false;
                    goto is_not_expanded;
                }
            }
        }
        m_is_expanded = true;
    }
    else
    {
    is_not_expanded:
        m_is_expanded = false;
    }
}
//-----------------------------------------------------------------------------
```



```cpp
Precalculated_buffer_parameters::Precalculated_buffer_parameters()
{
    clear();
}
//-----------------------------------------------------------------------------
void Precalculated_buffer_parameters::clear()
{
    Precalculated_pixel_format::clear();
    m_resolution = {0, 0};
    m_planes.clear();
    m_buffer_size = Bit_position(0);
}
//-----------------------------------------------------------------------------
void Precalculated_buffer_parameters::recalculate(
        const Pixel_format &format,
        const Vector<Unit::pixel> &resolution)
{
    Precalculated_pixel_format::recalculate(format);
    m_resolution = resolution;
    m_size_in_macropixels.set(
            resolution.x() / get_macropixel_size().x(),
            resolution.y() / get_macropixel_size().y());
    m_planes.resize(get_planes_count());
    Bit_position plane_offset = 0;
    for(int plane_index = 0; plane_index < get_planes_count(); plane_index++)
    {
        Plane_parameters &plane = m_planes[plane_index];
        plane.m_offset = plane_offset;
        const int rows_count = get_entry_rows_count_in_plane(plane_index);
        Bit_position bits_per_macropixel_row_in_plane = 0;
        for(int row_index = 0; row_index < rows_count; row_index++)
        {
            plane.m_rows[row_index].m_size =
                    m_size_in_macropixels.x()
                    * get_bits_per_entry_row_in_plane(plane_index, row_index);
            bits_per_macropixel_row_in_plane += plane.m_rows[row_index].m_size;
        }
        plane.m_size_per_row_of_macropixels = bits_per_macropixel_row_in_plane;
        plane.m_size =
                bits_per_macropixel_row_in_plane * m_size_in_macropixels.y();
        plane_offset += plane.m_size;
    }
    m_buffer_size = plane_offset;
}
```

```
#ifndef TCOMPONENTS_H
#define TCOMPONENTS_H

#include <vector>
#include <algorithm>
#include <yuv/utils.h>
#include <yuv/bit_position.h>
#include <yuv/coordinates.h>

struct Component
{
    /*
     * E'_X = K_XR * E'_R + K_XG * E'_G + K_XB * E'_B
     */
    double m_coeff[Rgba_component_rgb_count+1];
    double m_valid_range[2];
    double m_encoded_range[2];
};

inline bool operator==(const Component &a, const Component &b)
{
    return
            std::equal(
                a.m_coeff, a.m_coeff + Rgba_component_count + 1,
                b.m_coeff)
            && std::equal(
                a.m_valid_range, a.m_valid_range + 2,
                b.m_valid_range)
            && std::equal(
                a.m_encoded_range, a.m_encoded_range + 2,
                b.m_encoded_range);
}

typedef std::vector<Component> Color_space;

struct Entry
{
    Bit_position m_width;
};

struct Entry_row
{
    std::vector<Entry> m_entries;
};

struct Plane
{
    // one set of entries in plane corresponds to one macropixel
    std::vector<Entry_row> m_rows;
};

struct Component_coding
{
    int m_plane_index;
    int m_row_index;
    int m_entry_index;
};

struct Coded_pixel
{
    std::vector<Component_coding> m_components;
};
```



```
struct Macropixel_coding
{
    std::vector<Coded_pixel> m_pixels;
    Vector<Unit::pixel> m_size;
};

// TODO: add support for big-endian storage
// TODO: add support for rounding up and down in case
//       picture size is not integer multiplicity of macropixel size
struct Pixel_format
{
    std::vector<Plane> m_planes;
    Color_space m_components;
    Macropixel_coding m_macropixel_coding;
};

// full range RGB

const Color_space sRGB_color_space
{
    { // R
        {
            1,
            0,
            0
        },
        { 0, 1 },
        { 0, 1 }
    },
    { // G
        {
            0,
            1,
            0
        },
        { 0, 1 },
        { 0, 1 }
    },
    { // B
        {
            0,
            0,
            1
        },
        { 0, 1 },
        { 0, 1 }
    }
};

// studio RGB

const Color_space RGB_color_space
{
    { // R
        {
            1,
            0,
            0
        },
        { 0.0, 1.0 },
        { 16.0/256, 235.0/256 }
```

```
    },
    { // G
        {
            0,
            1,
            0
        },
        { 0.0, 1.0 },
        { 16.0/256, 235.0/256 }
    },
    { // B
        {
            0,
            0,
            1
        },
        { 0.0, 1.0 },
        { 16.0/256, 235.0/256 }
    }
};

// ITU 709 standard
const double ITU709_K_R = 0.2126;
const double ITU709_K_B = 0.0722;

const Color_space ITU709_YCbCr_color_space
{
    { // Y
        {
            ITU709_K_R,
            1 - ITU709_K_B - ITU709_K_R,
            ITU709_K_B
        },
        { 0.0, 1.0 },
        { 16.0/256, 235.0/256 }
    },
    { // Cb
        {
            -ITU709_K_R/(2*(1-ITU709_K_B)),
            -(1-ITU709_K_B-ITU709_K_R)/(2*(1-ITU709_K_B)),
            0.5
        },
        { -0.5, 0.5 },
        { 16.0/256, 240.0/256 }
    },
    { // Cr
        {
            0.5,
            -(1-ITU709_K_B-ITU709_K_R)/(2*(1-ITU709_K_R)),
            -ITU709_K_B/(2*(1-ITU709_K_R))
        },
        { -0.5, 0.5 },
        { 16.0/256, 240.0/256 }
    }
};

const Pixel_format yuv_420p_8bit
{
    { // planes
        { // plane Y
            { // rows
                { { { 8 }, { 8 } } },
```



```
                { { { 8 }, { 8 } } }
            }
        },
        { // plane U
            { // rows
                { { { 8 } } }
            }
        },
        { // plane V
            { // rows
                { { { 8 } } }
            }
        },
    },
    ITU709_YCbCr_color_space,
    { // macropixel coding
        { // coded pixels
            { // top left
                { { 0, 0, 0 }, { 1, 0, 0 }, { 2, 0, 0 } }
            },
            { // top right
                { { 0, 0, 1 }, { 1, 0, 0 }, { 2, 0, 0 } }
            },
            { // bottom left
                { { 0, 1, 0 }, { 1, 0, 0 }, { 2, 0, 0 } }
            },
            { // bottom right
                { { 0, 1, 1 }, { 1, 0, 0 }, { 2, 0, 0 } }
            }
        },
        { 2, 2 }
    }
};

class Precalculated_pixel_format
{
private:
    Pixel_format m_pixel_format;

    struct Entry_parameters
    {
        Bit_position m_offset;
        Coordinates<Unit::pixel, Reference_point::macropixel> m_sampling_point;
    };

    struct Entry_row_paramters
    {
        std::vector<Entry_parameters> m_entries;
        Bit_position m_bits_per_macropixel;
    };

    struct Plane_parameters
    {
        std::vector<Entry_row_paramters> m_rows;
        Bit_position m_bits_per_macropixel;
    };

    std::vector<Plane_parameters> m_planes;

    Bit_position m_bits_per_macropixel;
    bool m_is_expanded;
```

```
public:
    Precalculated_pixel_format();
    void clear();
    void recalculate(const Pixel_format &pixel_format);

    const Pixel_format &get_pixel_format() const
    {
        return m_pixel_format;
    }
    int get_components_count() const
    {
        return get_pixel_format().m_components.size();
    }
    int get_planes_count() const
    {
        return m_planes.size();
    }
    int get_entry_rows_count_in_plane(const int plane_index) const
    {
        return m_planes[plane_index].m_rows.size();
    }
    int get_entry_count_in_row_in_plane(
            const int plane_index,
            const int row_index) const
    {
        return m_planes[plane_index].m_rows[row_index].m_entries.size();
    }
    Bit_position get_bits_per_macropixel() const
    {
        return m_bits_per_macropixel;
    }
    Bit_position get_bits_per_macropixel_in_plane(const int plane_index) const
    {
        return m_planes[plane_index].m_bits_per_macropixel;
    }
    Bit_position get_bits_per_entry_row_in_plane(
            const int plane_index,
            const int row_index) const
    {
        return m_planes[plane_index].m_rows[row_index].m_bits_per_macropixel;
    }
    Bit_position get_bits_per_entry(
            const int plane_index,
            const int row_index,
            const int entry_index) const
    {
        return m_pixel_format.m_planes[
                plane_index].m_rows[
                row_index].m_entries[
                entry_index].m_width;
    }
    Bit_position get_bits_per_entry(
            const Coordinates<Unit::pixel, Reference_point::macropixel>
                &pixel_in_macropixel,
            const int component_index) const
    {
        const Component_coding &coding =
                m_pixel_format.m_macropixel_coding.m_pixels[
                get_pixel_coding_index(pixel_in_macropixel)].m_components[
                component_index];
        return
                m_pixel_format.m_planes[
```



```
                    coding.m_plane_index].m_rows[
                    coding.m_row_index].m_entries[
                    coding.m_entry_index].m_width;
    }
    Vector<Unit::pixel> get_macropixel_size() const
    {
        return m_pixel_format.m_macropixel_coding.m_size;
    }
    bool is_expanded() const
    {
        return m_is_expanded;
    }
private:
    int get_pixel_coding_index(
            const Coordinates<Unit::pixel, Reference_point::macropixel>
                &pixel_in_macropixel) const
    {
        return
                pixel_in_macropixel.y() * get_macropixel_size().x()
                + pixel_in_macropixel.x();
    }
};

class Precalculated_buffer_parameters : public Precalculated_pixel_format
{
public:
    Precalculated_buffer_parameters();
    void recalculate(
            const Pixel_format &format,
            const Vector<Unit::pixel> &resolution);
    void clear();

    const Vector<Unit::pixel> &get_resolution() const
    {
        return m_resolution;
    }
    Bit_position get_buffer_size() const
    {
        return m_buffer_size;
    }
    Bit_position get_plane_size(int plane_index) const
    {
        return m_planes[plane_index].m_size;
    }
    Bit_position get_plane_offset(int plane_index) const
    {
        return m_planes[plane_index].m_offset;
    }
    Bit_position get_macropixel_row_in_plane_size(int plane_index) const
    {
        return m_planes[plane_index].m_size_per_row_of_macropixels;
    }
    Vector<Unit::macropixel> get_size_in_macropixels() const
    {
        return m_size_in_macropixels;
    }
    Bit_position get_entry_offset(
            const Coordinates<Unit::pixel, Reference_point::picture>
                &coordinates,
            const int component_index) const
    {
        throw "TODO";
```

```
        return Bit_position(0);
    }
    using Precalculated_pixel_format::get_bits_per_entry;
    Bit_position get_bits_per_entry(
            const Coordinates<Unit::pixel, Reference_point::picture>
                &coordinates,
            const int component_index) const
    {
        Coordinates<Unit::pixel, Reference_point::macropixel>
                pixel_in_macropixel;
        cast_to_macropixels(
                coordinates,
                get_macropixel_size(),
                pixel_in_macropixel);
        return get_bits_per_entry(pixel_in_macropixel, component_index);
    }
    Coordinate_range<Unit::pixel, Reference_point::picture> get_pixel_range(
            ) const
    {
        return Coordinate_range<Unit::pixel, Reference_point::picture>(
                Coordinates<Unit::pixel, Reference_point::picture>(0, 0),
                m_resolution);
    }

private:
    Vector<Unit::pixel> m_resolution;
    Vector<Unit::macropixel> m_size_in_macropixels;

    struct Entry_row_parameters
    {
        Bit_position m_size;
    };

    struct Plane_parameters
    {
        std::vector<Entry_row_parameters> m_rows;
        Bit_position m_size_per_row_of_macropixels;
        Bit_position m_size;
        Bit_position m_offset;
    };

    std::vector<Plane_parameters> m_planes;

    Bit_position m_buffer_size;
};

Pixel_format get_expanded_pixel_format(
        const std::vector<Component> &components,
        const std::vector<Entry> &entries);
Pixel_format get_expanded_pixel_format(const Pixel_format &input);

#endif // TCOMPONENTS_H
```

```
#include <yuv/TYUV_file.h>
#include <yuv/Errors.h>

#include <iostream>

Yuv_file::Yuv_file() :
    m_parameters_valid(false)
{
    m_file.exceptions(std::ios_base::badbit | std::ios_base::failbit);
}
//------------------------------------------------------------------------------
Yuv_file::Yuv_file(const boost::filesystem::path &path) :
    m_path(path),
    m_parameters_valid(false)
{
    m_file.exceptions(std::ios_base::badbit | std::ios_base::failbit);
    open(path);
}
//------------------------------------------------------------------------------
bool Yuv_file::is_open() const
{
    return m_file.is_open();
}
//------------------------------------------------------------------------------
void Yuv_file::open(const boost::filesystem::path &path)
{
    std::cout << "TYUV_file::TYUV_file(const std::string file_name)\n";
    m_file.open( path );
    init_file_parameters();
}
//------------------------------------------------------------------------------
void Yuv_file::close()
{
    m_file.close();
}
//------------------------------------------------------------------------------
void Yuv_file::set_pixel_format(const Pixel_format &pixel_format)
{
    m_parameters_valid = false;
    m_pixel_format = pixel_format;
}
//------------------------------------------------------------------------------
const Pixel_format &Yuv_file::get_pixel_format() const
{
    return m_pixel_format;
}
//------------------------------------------------------------------------------
void Yuv_file::set_resolution(const Vector<Unit::pixel> &resolution)
{
    m_parameters_valid = false;
    m_resolution = resolution;
}
//------------------------------------------------------------------------------
Vector<Unit::pixel> Yuv_file::get_resolution() const
{
    return m_resolution;
}
//------------------------------------------------------------------------------
Bit_position Yuv_file::get_frame_size()
{
    recalculate_parameters();
    return m_buffer_parameters.get_buffer_size();
```



```
}
//------------------------------------------------------------------------------
int Yuv_file::get_frames_count()
{
    return Bit_position(m_file_size, 0) / get_frame_size();
}
//------------------------------------------------------------------------------
Picture_buffer Yuv_file::extract_buffer(
        int picture_number,
        const Coordinates<Unit::pixel, Reference_point::picture> &start,
        const Coordinates<Unit::pixel, Reference_point::picture> &end)
{
    const Vector<Unit::pixel> macropixel_size =
            m_buffer_parameters.get_macropixel_size();
    check_range(0, picture_number, get_frames_count());
    check_range(0, start.x(), end.x());
    check_range(0, start.y(), end.y());
    check_range(0, end.x(), m_resolution.x() + 1);
    check_range(0, end.y(), m_resolution.y() + 1);
    Coordinates<Unit::pixel, Reference_point::macropixel> dummy_vector;
    my_assert(
            start
            == cast_to_pixels(
                cast_to_macropixels(start, macropixel_size, dummy_vector),
                macropixel_size),
            "non-integer macropixel boundaries are not supported");
    my_assert(
            end
            == cast_to_pixels(
                cast_to_macropixels(end, macropixel_size, dummy_vector),
                macropixel_size),
            "non-integer macropixel boundaries are not supported");

    recalculate_parameters();

    const Pixel_format &pixel_format = get_pixel_format();
    const Vector<Unit::pixel> buffer_size = end - start;
    Picture_buffer buffer;
    buffer.allocate(buffer_size, pixel_format);
    Bit_position offset_in_buffer = 0;
    const Vector<Unit::macropixel> buffer_size_in_macropixels =
            buffer.get_parameters().get_size_in_macropixels();
    const Vector<Unit::macropixel> picture_size_in_macropixels =
            m_buffer_parameters.get_size_in_macropixels();
    const Bit_position picture_size = get_frame_size();
    const Bit_position picture_offset = picture_number * picture_size;

    const int planes_count = pixel_format.m_planes.size();
    for(int plane_idx = 0; plane_idx < planes_count; plane_idx++)
    {
        Bit_position plane_offset =
                m_buffer_parameters.get_plane_offset(plane_idx);
        const int pixel_rows_in_macropixel_in_plane =
                m_buffer_parameters.get_entry_rows_count_in_plane(plane_idx);
        const Bit_position macropixel_row_in_plane_size_in_bits =
                m_buffer_parameters.get_macropixel_row_in_plane_size(plane_idx);
        for(int buffer_macropixel_row = 0;
            buffer_macropixel_row < buffer_size_in_macropixels.y();
            buffer_macropixel_row++)
        {
            const int macropixel_row =
                    cast_to_macropixels(
```

```
                        start,
                        macropixel_size,
                        dummy_vector).y()
                    + buffer_macropixel_row;
            const Bit_position macropixel_row_offset =
                    macropixel_row_in_plane_size_in_bits * macropixel_row;
            Bit_position row_in_macropixel_offset = 0;
            for(int row_in_macropixel = 0;
                row_in_macropixel < pixel_rows_in_macropixel_in_plane;
                row_in_macropixel++)
            {
                const Bit_position bits_per_entry_row_in_plane =
                        m_buffer_parameters.get_bits_per_entry_row_in_plane(
                            plane_idx, row_in_macropixel);
                const Bit_position x_position_offset =
                        cast_to_macropixels(
                            start,
                            macropixel_size,
                            dummy_vector).x()
                        * bits_per_entry_row_in_plane;
                const Bit_position read_length =
                        buffer_size_in_macropixels.x()
                        * bits_per_entry_row_in_plane;
                const Bit_position offset =
                        picture_offset
                        + plane_offset
                        + macropixel_row_offset
                        + row_in_macropixel_offset
                        + x_position_offset;

                my_assert(
                        offset.get_bits() == 0
                        && offset_in_buffer.get_bits() == 0,
                        "TODO: handle non-byte boundary reads");

                m_file.seekg(offset.get_bytes());
                m_file.read(reinterpret_cast<char *>(buffer.get_data().data() +
                        offset_in_buffer.get_bytes()),
                        read_length.get_bytes());

                row_in_macropixel_offset +=
                        bits_per_entry_row_in_plane *
                        picture_size_in_macropixels.x();
                offset_in_buffer += read_length;
            }
        }
    }
    return buffer;
}
//----------------------------------------------------------------------------------
void Yuv_file::init_file_parameters()
{
    // get the file size
    {
        m_file.seekg(0, std::ios_base::beg);
        auto begin = m_file.tellg();
        m_file.seekg(0, std::ios_base::end);
        auto end = m_file.tellg();
        m_file_size = end - begin;
    }
    std::cout << "m_file_size = " << m_file_size << std::endl;
    m_parameters_valid = false;
```



```
}
//----------------------------------------------------------------------------------
void Yuv_file::recalculate_parameters()
{
    if(!m_parameters_valid)
        m_buffer_parameters.recalculate(m_pixel_format, m_resolution);
    m_parameters_valid = true;
}
```

```
#ifndef TYUV_FILE_H
#define TYUV_FILE_H

#include <string>
#include <fstream>
#include <array>
#include <boost/filesystem.hpp>
#include <boost/filesystem/fstream.hpp>

#include <yuv/utils.h>
#include <yuv/TComponents.h>
#include <yuv/picture_buffer.h>

class Yuv_file
{
public:
    Yuv_file();
    Yuv_file(const boost::filesystem::path &path);
    bool is_open() const;
    void open(const boost::filesystem::path &path);
    void close();
    void drawFrameGL() const;
    void set_pixel_format(const Pixel_format &pixel_format);
    const Pixel_format &get_pixel_format() const;
    void set_resolution(const Vector<Unit::pixel> &resolution);
    Vector<Unit::pixel> get_resolution() const;
    Bit_position get_frame_size();
    int get_frames_count();
    Picture_buffer extract_buffer(
            int picture_number,
            const Coordinates<Unit::pixel, Reference_point::picture> &start,
            const Coordinates<Unit::pixel, Reference_point::picture> &end);

private:
    void init_file_parameters();
    void recalculate_parameters();

    boost::filesystem::path m_path;
    boost::filesystem::ifstream m_file;
    size_t m_file_size;
    Pixel_format m_pixel_format;
    Precalculated_buffer_parameters m_buffer_parameters;
    bool m_parameters_valid;
    Vector<Unit::pixel> m_resolution;
};

#endif // TYUV_FILE_H
```

```
#ifndef UTILS_H
#define UTILS_H

#include <cstdint>

typedef std::uint8_t Byte;
const int bits_in_byte = 8;
//-----------------------------------------------------------------------------
enum Rgba_component
{
    Rgba_component_red,
    Rgba_component_green,
    Rgba_component_blue,
    Rgba_component_alpha,
    Rgba_component_count,
    Rgba_component_rgb_count = 3
};
//-----------------------------------------------------------------------------
template<typename TNumber>
TNumber round_down(TNumber value,TNumber divisor)
{
    return value - value%divisor;
}
//-----------------------------------------------------------------------------
template<typename TNumber>
TNumber round_up(TNumber value, TNumber divisor)
{
    return round_down(value + divisor - static_cast<TNumber>(1), divisor);
}

#endif // UTILS_H
```