DH2323 LAB 1

In this first lab you will familiarize yourself with the lab environment, C++ and the libraries SDL and GLM, and how to use it to do graphics programming. This lab consists of three parts:

1. Set-up of the lab environment.

2. Introduction to 2D computer graphics. This part covers image representation, colors, pixels and linear interpolation. You should write a program that draws the image seen to the left in Figure 1.

3. Introduction to 3D computer graphics. This part covers the pinhole camera and how it projects 3D points to 2D. You should write a program that produces a starfield effect, i.e. lots of 3D points moving towards you, as seen below in Figure 1.

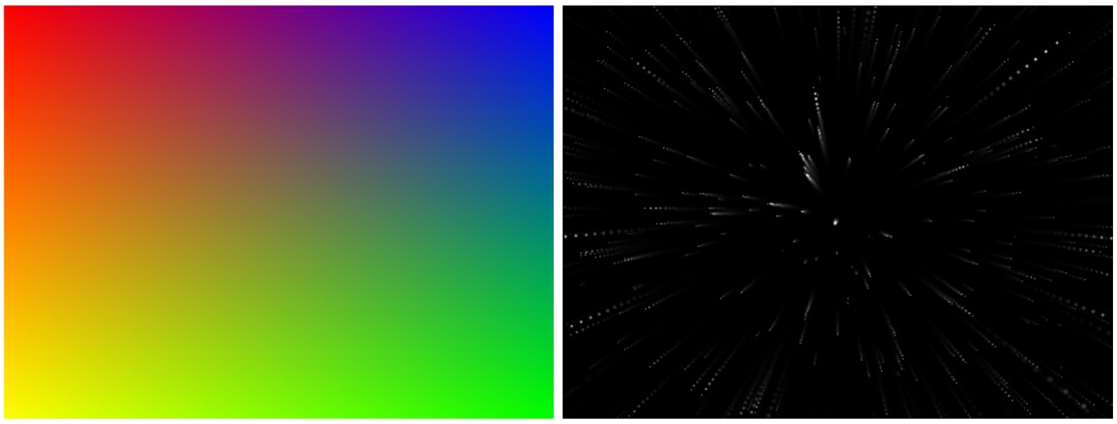

Figure 1: The output from this lab.

1 Setup of the Lab Environment

You have a number of choices for how to do the labs:

1.1. use computers provided by KTH.
1.2. use your own computer the way you prefer.
1.3. run a Virtual Machine on your computer.

The project uses two external libraries. Here are the links to the websites where you can find the documentation:

SDL: http://www.libsdl.org/
GLM: http://glm.g-truc.net/

First of all download the latest version of the project files from the course website. The zip file includes a simple C++ skeleton and the GLM library.
All the lab assignments come with a *Cmake* file to make the compilation easy and independent from the platform you desire to use. You just need a few steps to set your machine up correctly. Those will be explained later in detail in this document.

Important: Make sure to include all of the project and auxiliary files that will allow your lab work to be compiled.

1.1 KTH Computers

The computers have SDL already installed and GLM comes with the zip file you downloaded. You just need to open the terminal go to the path where you downloaded your files and run:

```
-> cmake .
-> make
```

1.2 Your System

You will need to download all the libraries and set some environment variables in order to use the cmake file included in the project folder. We use these libraries:

SDL: http://www.libsdl.org/
GLM: http://glm.g-truc.net/

The cmake file included can automatically find SDL and you only need to download and install it. GLM is already included in the project zip file. If you do not have cmake already installed in your system please download it from: http://www.cmake.org/

1.3 Virtual Machine

You can run a virtual machine. To run it you need to download and install VirtualBox that can be found at: https://www.virtualbox.org/.
If you have problems setting up a virtual machine, please contact the course responsible and we will attempt to provide one for you!

1.4 Test

In order to check that everything works correctly please download the project files from the course website, open a terminal and go into your project folder. Then run CMake and if everything is set up correctly you should not receive any error. Run make to generate an executable from the project files.

2 Introduction to 2D Computer Graphics

In all the labs of this course you will use the library SDL to draw pixels on the screen. SDL is quite easy to use and we provide a skeleton program that initializes the library and functions for drawing pixels etc. Thus, you probably do not need to look into the SDL documentation, but if you are interested we recommend you to have a look at:

http://www.libsdl.org/cgi/docwiki.cgi

It is a very good library well worth learning. References for C++ and its standard library can be found at:

http://www.cplusplus.com
http://www.cppreference.com

We provide a skeleton program, shown below, which you can use as a base for the labs. It uses SDL to create a surface which you can draw on and then fills it with a constant color:

```
#include "SDL.h"
#include <iostream>
#include <glm/glm.hpp>
#include <vector>
#include "SDLauxiliary.h"
using namespace std;
using glm::vec3;

// ---------------------------------------------------------
// GLOBAL VARIABLES
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
SDL_Surface* screen;

// ---------------------------------------------------------
// FUNCTION DECLARATIONS
void Draw();

// ---------------------------------------------------------
// FUNCTION DEFINITIONS
int main( int argc, char* argv[] )
{
      screen = InitializeSDL( SCREEN_WIDTH, SCREEN_HEIGHT );
      while( NoQuitMessageSDL() )
      {
            Draw();
      }
      SDL_SaveBMP( screen, "screenshot.bmp" );
      return 0;
}

void Draw()
{
      if( SDL_MUSTLOCK(screen) )
      SDL_LockSurface(screen);
      for( int y=0; y<SCREEN_HEIGHT; ++y )
      {
            for( int x=0; x<SCREEN_WIDTH; ++x )
            {
                  vec3 color(0,0,1);
                  PutPixelSDL( screen, x, y, color );
            }
```

```
	}
	if( SDL_MUSTLOCK(screen) )
		SDL_UnlockSurface(screen);

	SDL_UpdateRect( screen, 0, 0, 0, 0 );
}
```

This program uses SDL to draw every pixel of the screen blue, by manually looping through all pixels. There are some helper functions that simplify the use of SDL:

InitializeSDL
This function takes the width and the height of the surface we want to create as arguments. It initializesW SDL and returns a pointer to the created surface. This surface represents the video memory. We can then send this surface pointer to PutPixel when we want to draw a pixel.

NoQuitMessageSDL
This function returns true as long as SDL does not receive any message to quit the program. This will happen if e.g. the user presses the cross in the corner of the window. Pressing the Esc-key will also cause this function to return false.

PutPixelSDL
This function is used to draw pixels. As arguments it takes a pointer to a surface as well as the position (x; y) and color of the pixel to draw. The color is represented by a vector, glm::vec3, with the red, green and blue components of the color. The color components are clamped to be between zero and one. If you are interested you can find the code for these helper functions in the header file SDLauxiliary.h. Before drawing anything by calling PutPixel, the video memory needs to be locked as shown in the example code. The for-loops then loop over all rows and columns of the screen and set the pixels to blue. After the drawing is done the screen needs to be unlocked and you also need to call SDLUpdateRect to update the surface.

2.1 Color the Screen

Experiment with drawing different colors until you understand the color model. You can try to fill the screen with: red, yellow, green, cyan, blue, magenta, black, white and different intensities like dark red or pink.

2.2 Linear Interpolation

Your final task in this part of the lab is to fill the surface with the rainbow effect seen to the left in Figure 1. You should specify a color for each corner of the surface and then interpolate these in between. Linear interpolation is often used in computer graphics programming. In the rasterization lab you will use it a lot. That is why we want you to implement some simple linear interpolation already in this first lab. You should write a function that does this:

```
void Interpolate( float a, float b, vector<float>& result );
```

Extend the skeleton program by writing this function. The std::vector result should be filled with values linearly interpolated between *a* and *b*. The size of the std::vector should have been set before calling the function, which should only fill it with values. The Interpolate-function should be used like this:

```
vector<float> result( 10 ); // Create a vector width 10 floats
Interpolate( 5, 14, result ); // Fill it with interpolated values
for( int i=0; i<result.size(); ++i )
      cout << result[i] << " "; // Print the result to the terminal
```

This should produce the output:

```
5 6 7 8 9 10 11 12 13 14
```

You might need to treat the special case when the size of the vector is 1 with an if-statement, to avoid division with zero. Think about a suitable value to return in this case. You can write the test code above in the beginning of the main function. After you got the Interpolation working for should make a new version that works for glm::vec3:

```
void Interpolate( vec3 a, vec3 b, vector<vec3>& result );
```

To test that your implementation produces the right output you can try to use it like this:

```
vector<vec3> result( 4 );
vec3 a(1,4,9.2);
vec3 b(4,1,9.8);
Interpolate( a, b, result );
for( int i=0; i<result.size(); ++i )
{
      cout << "( "
      << result[i].x << ", "
      << result[i].y << ", "
      << result[i].z << " ) ";
}
```

This should produce the output:

```
( 1, 4, 9.2 ) ( 2, 3, 9.4 ) ( 3, 2, 9.6 ) ( 4, 1, 9.8 )
```

Now you have a function that can be used to interpolate the most important quantities in computer graphics: 3D positions and colors, which we will both represent with glm::vec3.

2.3 Bilinear Interpolation of Colors

Now that your interpolation works you should use it in the Draw-function to interpolate colors across the screen. First define the colors for each corner of the screen:

```
vec3 topLeft(1,0,0); // red
vec3 topRight(0,0,1); // blue
vec3 bottomLeft(0,1,0); // green
vec3 bottomRight(1,1,0); // yellow
```

You should then interpolate linearly between these colors to get a color for each pixel of the screen. This can be done by first interpolating the values of the very left, and right side of the screen:

```
vector<vec3> leftSide( SCREEN_HEIGHT );
vector<vec3> rightSide( SCREEN_HEIGHT );
Interpolate( topLeft, bottomLeft, leftSide );
Interpolate( topRight, bottomRight, rightSide );
```

Then for each row of the screen you can interpolate between the values of the left and right side of this row. This can be done in the for-loop in the Draw-function. Try to get the result shown in Figure 1. After succeeding in this you can experiment with different colors of the corners to get some more intuition about what happens when colors are mixed/interpolated.

3 Starfield

In the previous part of the lab you have practiced interpolation and working with colors. In this part we will move ahead and start looking at geometry and in specific projections. We will also look at how we can update parameters within the framework to create an animated image. In specific we will create an implementation of the classic star field effect which have been used for things such as screen-blankers and famously for showing the Millennium Falcon's traveling through space. The effect is created by modeling a set of 3D points that moves through space (in time) towards the camera. If the camera is pointing along the z-axis then this will be the update equations for a point $p^i = (x^{i,} y^i, z^i)$:

$$\begin{aligned} x_t^i &= x_{t-1}^i \\ y_t^i &= y_{t-1}^i \\ z_t^i &= z_{t-1}^i - v \cdot dt \end{aligned} \tag{1}$$

where *v* is the velocity of the points and *dt* is the elapsed time between the two frames. To be able to use the update equations we need to initialize the points and determine the locations at time zero. Create a new global variable to store the locations of all stars:

```
vector<vec3> stars( 1000 );
```

In this case we have created one thousand stars, represented as 3D points. Next you should initialize the location of every star at the first frame. If we assume that stars are randomly located in space we can use the random number generator to initialize the locations:

```
float r = float(rand()) / float(RAND_MAX);
```

This will produce a random number between zero and one. Create a for-loop in the beginning of the main function that loops through all stars and sets random positions within:

$$\begin{aligned} -1 \leq x \leq 1 \\ -1 \leq y \leq 1 \\ 0 < z \leq 1 \end{aligned} \tag{2}$$

3.1 Projection - Pinhole Camera

Now when we have a created the set of 3D points we need to implement the rendering of the points on the screen. In order to do this we need to implement a simple pinhole camera that will provide the perspective effect when the points are moving towards the camera. To make things simple we assume that the camera is positioned at the origin and looking in the direction of the z-axis. We assume that the x-axis and y-axis of the 3D coordinate system points in the same directions as those for the 2D screen coordinate system. However, we assume that the center of the 3D coordinate system is at the center of the screen having width W and height H. The projection from a 3D point (x^i, y^i, z^i) to a 2D screen coordinate (u^i, v^i) can then be written:

$$\begin{aligned} u^i &= f \cdot \frac{x^i}{z^i} + \frac{W}{2} \\ v^i &= f \cdot \frac{y^i}{z^i} + \frac{H}{2}. \end{aligned} \tag{3}$$

where f is the focal length of the camera. A good value for it is **f=H/2**. Then the vertical field of view for the camera will be 90 degrees. What is the resulting horizontal field of view? Try to calculate this.

Implement the projection and drawing of all points in the Draw function. Before drawing the projected position/pixel of the stars as white with PutPixelSDL you should make the background black. In the beginning of this lab we filled the whole screen with a single color by explicitly looping through all pixels and calling PutPixelSDL. A faster way to fill the whole screen with a black color is to call:

```
SDL_FillRect( surface, 0, 0 );
```

You can read more about this function in the SDL documentation. It should be called before locking the surface. Your implementation of the Draw function should look something like:

```
void Draw()
{
	SDL_FillRect( screen, 0, 0 );
	if( SDL_MUSTLOCK(screen) )
	SDL_LockSurface(screen);
	for( size_t s=0; s<stars.size(); ++s )
	{
		// Add code for projecting and drawing each star
	}
	if( SDL_MUSTLOCK(screen) )
	SDL_UnlockSurface(screen);
	SDL_UpdateRect( screen, 0, 0, 0, 0 );
}
```

If you do this you should see a black sky with some white stars on the screen.

3.2 Motion

It is now time to make things a bit more interesting by adding motion. We recommend that you create a new function:

```
void Update();
```

to handle this. This function should be called each frame before the drawing, i.e. the main loop of the program should look like:

```
while( NoQuitMessageSDL() )
{
	Update();
	Draw();
}
```

To measure the time between two frames you can use the function SDLGetTicks. This function returns the number of milliseconds since SDL was initialized.
First create a global variable:

```
int t;
```

Before entering the main-loop you can initialize t to hold the current time:

```
t = SDL_GetTicks();
```

Then in the Update function you can compute the time since the last update by writing:

```
void Update()
{
      int t2 = SDL_GetTicks();
      float dt = float(t2-t);
      t = t2;
}
```

The local variable dt will then hold the number of milliseconds that has passed since Update was called last time. Use this and write code that updates the position of all stars according to *equation 1*. Choose a reasonable value for the velocity v. If the stars are outside of the volume specified in *equation 2* after the update they should be wrapped around to the other side, to make the illusion that the stars never end no matter far we travel:

```
for( int s=0; s<stars.size(); ++s )
{
      // Add code for update of stars
      if( stars[s].z <= 0 )
      stars[s].z += 1;
      if( stars[s].z > 1 )
            stars[s].z -= 1;
}
```

If you have implemented the code above correctly, including moving the stars according to their velocity, you should see the stars moving towards you. You might notice a popping effect when the stars wrap around from the nearest to the farthest part of the volume. To make the transition smoother you can fade the brightness of the stars, i.e. let it depend on the distance to the camera. A physically motivated way to do this is to let the brightness be inversely proportional to the squared distance:

```
vec3 color = 0.2f * vec3(1,1,1) / (stars[s].z*stars[s].z);
```

This should be all the necessary steps to create the starfield effect. In order to pass this part of the lab you should show an implementation with a set of points moving towards the screen. Further, you should be able to explain and derive all the equations above.
If you think this lab was way to easy you can also try to add motion blur to the starfield. That will look even cooler.