COMS30115 - Coursework 1^0

Carl Henrik Ek & Magnus Burenius

January 24, 2019

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

- Setup of the lab environment.
- 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.
- 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 to the right in Figure 1.

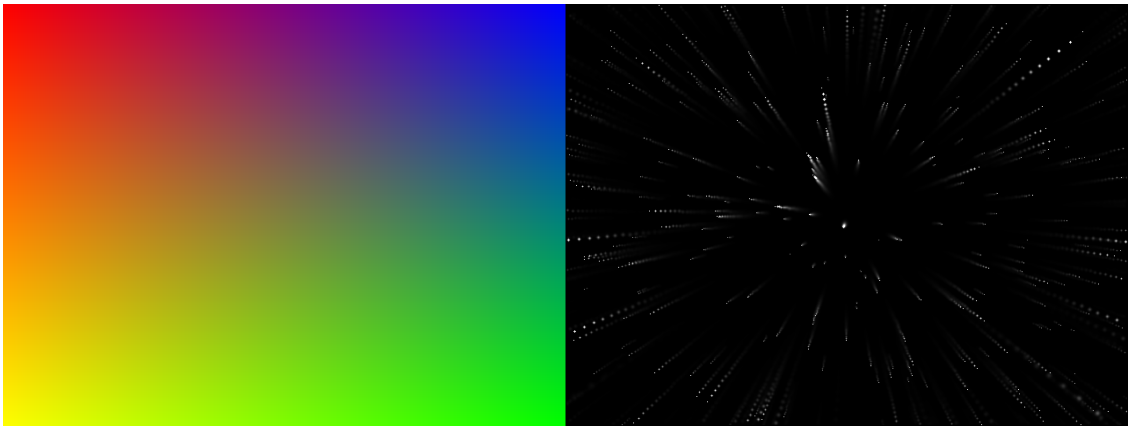

Figure 1: The output from this lab

1 Setup of the Lab Environment

We have tried to use an environment that is fairly agnostic to the underlying architecture so it should be fairly straight-forward to get running on most machines. 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`

The idea of this unit is to make sure that we write everything ourselves, therefore you are most likely not going to have much use for the `SDl` documentation. With `GLM` the story is slightly different, you will use a lot of Linear Algebra routines which are already implemented in this library. However, writing your own math library is not a bad idea at all.

First clone the repository of the unit by writing the following in your terminal,

Code

```
git clone https://github.com/carlhenrikek/COMS30115.git
```

You should now have a directory called `COMS30115` where all the course material will be availible. Now execute the following series of commands,

Code

```
cd COMS30115/Labs/starfield/
make
./Build/skeleton
```

If everything works as it should you should now have opened a small window that generates some random pattern.

We will test your assignment on the lab machines so you are not allowed to use any platform dependent function or any other library from the ones listed in the assignment. Furthermore if you use an IDE (i.e. Visual Studio, Eclipse, XCode, ...) make sure that your project files can be compiled without it. The project should be submitted with a `README` and a `Makefile`. The only thing we should need to do in order to run the code is to repeate the proceedure above[1].

[1] In the later labs some of you might want to use some external libraries, before you do so verify that with us and then make sure that you provide clear instructions on how they should be initialised.

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`

References for C++ and its standard library can be found at:

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

If you are interested to work with computer graphics in any form or way, you will really want to become a `C Ninja`, *C is the best assembler* on the market and to write efficient code you need to know what code your compiler actually generates, i.e. you think in assembly but write in C. 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 random placed pixels,

Code

```
#include <iostream>
#include <glm/glm.hpp>
#include <SDL.h>
#include "SDLauxiliary.h"
#include "TestModel.h"
#include <stdint.h>

using namespace std;
using glm::vec3;
using glm::mat3;

#define SCREEN_WIDTH 320
#define SCREEN_HEIGHT 256
#define FULLSCREEN_MODE false
```

The above code simply includes the basic libraries that we will use, `SDLauxilary.h` contains a couple of functions that we have written and `TestModel.h` contains the vertex data that we will be using for the coursework. We will use `glm`'s vector and matrix structs a lot so we include them directly. The last couple of lines are just some useful defines as you'll need to use them globally a lot and we want to avoid global variables.[2]

[2]The resolution `320x256` is low-res PAL on a Commodore Amiga.

Code

```
/* ------------------------------------------------------------------------------*/
/* FUNCTIONS                                                                     */

void Update();
void Draw(screen* screen);

int main( int argc, char* argv[] )
{

  screen *screen = InitializeSDL( SCREEN_WIDTH, SCREEN_HEIGHT, FULLSCREEN_MODE );

  while( NoQuitMessageSDL() )
    {
      Update();
      Draw(screen);
      SDL_Renderframe(screen);
    }

  SDL_SaveImage( screen, "screenshot.bmp" );

  KillSDL(screen);
  return 0;
}
```

The above code is the `main` function that will be called first. The first call is to a utillity function to open up a screen for us to draw on.

InitializeSDL This function takes the width and the height of the surface we want to create as arguments. It initializes SDL and returns a pointer to a screen `struct`. This `struct` will be the way we interact with SDL and it contains the following elements,

Code

```
typedef struct{
  SDL_Window *window;
  SDL_Renderer *renderer;
  SDL_Texture *texture;
  int height;
  int width;
  uint32_t *buffer;
} screen;
```

`SDL_Window` is the actual window that you render to, one of the nice things with SDL2 compared to 1.2 is that you can render to multiple windows so if you need you can add more. `SDL_Renderer` is the object that actually renders the window, in the current setup the rendering is synced to your vertical blanking interput so you will never render faster than the refresh rate of your screen. `SDL_Texture` is a memory struct on your GPU and this is what will be rendered to the window. `buffer` is the memory where we will actually draw our pixels. The motivation for not drawing directly into the texture is to avoid context switches (when the computer stalls transfering between RAM and VRAM). We therefore draw in `buffer` and then copy the whole memory region to `texture` in a single pass.

Once the screen has been opened we enter a loop based on the state of `NoQuitMessageSDL`.

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.

You are most likely going to keep this loop very much as it is through the unit, it has calls to three functions `Update`, `Draw` and `SDL_Renderframe`.

SDL_Renderframe This function copies `buffer` to `texture` and takes care of the double-buffering to avoid glitches in our rendering.

Code

```
/*Place your drawing here*/
void Draw(screen* screen)
{
  /* Clear buffer */
  memset(screen->buffer, 0, screen->height*screen->width*sizeof(uint32_t));

  vec3 colour(1.0,0.0,0.0);
  for(int i=0; i<1000; i++)
    {
      uint32_t x = rand() % screen->width;
      uint32_t y = rand() % screen->height;
      PutPixelSDL(screen, x, y, colour);
    }
}

/*Place updates of parameters here*/
void Update()
{
  static int t = SDL_GetTicks();
  /* Compute frame time */
  int t2 = SDL_GetTicks();
  float dt = float(t2-t);
  t = t2;
  /*Good idea to remove this*/
  std::cout << "Render time: " << dt << " ms." << std::endl;
  /* Update variables*/
}
```

The `Update` functions is a good place to add things that should change between frames, camera etc. While the `Draw` function is the place where we will do the actual rendering of the image. In this case the update function is simply calculating the time between two consequitive frames using a `static` variable. Importantly the print statement will take an awful lot of time to do so I suggest that you make something that you can `set/reset` with a parameter. The `Draw` function first clears the screen[3] and then creates a pixel value that it draws at some random places on the screen using the function,

PutPixelSDL This function is used to draw pixels. As arguments it takes a pointer to a `screen struct` 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 during the rendering pass.

[3] Try running the code without executing this.

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 next 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:

Code

```
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 st::vector should have been set before calling the function, which should only fill it with values. The Interpolate-function should be used like this:

Code

```
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:

Code

```
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 float you should make a new version that works for `glm::vec3`:

Code

```
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:

Code

```
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:

Code

```
( 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:

Code

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

Code

```
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 **??**. 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 *starfield* 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 variable to store the locations of all stars:

Code

```
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:

Code

```
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,

Code

```
memset(screen->buffer, 0, width*height*sizeof(uint32_t));
```

Your implementation of the Draw function should look something like:

Code

```
void Draw()
{
  memset(screen->buffer, 0, width*height*sizeof(uint32_t));
  for( size_t s=0; s<stars.size(); ++s )
    {
      /* Add code for projecting and drawing each star */
    }

}
```

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 extend the `Update()` function to handle this. This function is be called each frame before the drawing, i.e. the main loop of the program,

Code

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

As we do not have access to the hardware interupts on the computer the function will be called at different frequency dependent on how long it takes to calculate a frame. Therefore we need to measure the time between calls to make updates that are consistent. To measure the time between two frames you can use the function `SDL_GetTicks`. This function returns the number of milliseconds since SDL was initialized. As we need to remember the time from the previous call to the `Update` function we create a `static` variable that won't be trashed when we leave its scope. Then in the Update function you can compute the time since the last update by writing:

Code

```
void Update()
{
  static int t = SDL_GetTicks();
  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 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:

Code

```
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:

Code

```
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 want to add stuff to your starfield you can think of things like motion-blur and moving both the stars and the camera. If you read the SDL documentation it is rather easy to read input from mouse of keyboard so you can use this to control something and make the whole thing interactive. Play around, being creative is half of computer graphics.

4 The End

If you have reached this part you are done with the first part of the lab, now go back and read the abstract again. Do you feel that you have a good grasp of the points that are outlined there? If so, well done, if not, have a look through what you have done again and see if you can connect the dots. It is now time to move on to the real coursework and start implementing a raytracer.