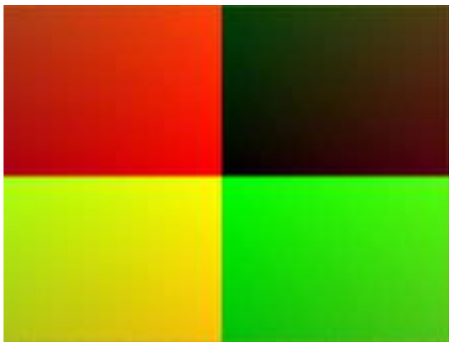
Advanced Graphics Programming

Workshop One (Part A) – Raycasting (3D)

The project you will start today and develop over subsequent workshops is developing the skills required for Assignment 1. This workshop is based on the first step in ray casting: constructing rays from the camera that pass through each pixel and when completed should look something similar to this:



Part 1: Setup

Step 1: Add an existing vector class to a new Visual Studio C++ project (Win32 Console Application). The ray casting algorithm is based on vector arithmetic so the first step is to add an existing vector class, either your own or a 3rd party library. I can recommend the vec3 class in the GLM library which is already on the lab PCs in C:\OpenGL\ glm-0.9.6.3 folder and can also be downloaded from Moodle.

GLM is a header only library, so there is nothing to build. To use it you should copy the glm-0.9.6.3 folder from the C:\OpenGL folder to your working directory for Visual Studio, then in the new project properties set the C/C++ -> Additional Include Directories to the relative path to the glm folder e.g. ..\..\..\glm-0.9.6.3. Finally include <glm/glm.hpp> in the C++ file where you want to use the vec3 (see example code snippet below).

```
#include <iostream>
#include <glm/glm.hpp>

using namespace std;
using namespace glm;

void step1()
{
	// Add two vectors together and output the result on the console window
	vec3 v1(1, 2, 3);
	vec3 v2(3, 4, 5);

	vec3 v3 = v1 + v2;

	cout << "v1 " << "[" << v1.x << " " << v1.y << " " << v1.z << "]" << endl;
	cout << "v2 " << "[" << v2.x << " " << v2.y << " " << v2.z << "]" << endl;
	cout << "v3 " << "[" << v3.x << " " << v3.y << " " << v3.z << "]" << endl;
}
```

For further information on glm the core functions are listed here: http://glm.g-truc.net/0.9.2/api/a00239.html#ga07ff16965f11fa17122ac874ed492276

Step 2: Setup the view plane. The view plane is a 2D image of the scene and needs to be stored in memory so that you can display it. A simple approach is to create a two-dimensional array based on the width and height of your image (e.g. 640 x 480). This makes setting the colour for each pixel very simple.

Create a two dimensional array of size width by height.

```
        vec3 image[WIDTH][HEIGHT];
```

Warning: a regular array declaration like the example above for small arrays but you will need to use a pointer array format as described here: http://www.dreamincode.net/forums/topic/144007-multidimensional-arrays/ for the image otherwise you may get a runtime exception.

Temporarily, initialise **each pixel** in your image array to the colour yellow. The colour of each pixel should be represented by 3 values (R, G, B) where R, G and B should be set to values between 0 and 1. You will need to use the pointer format rather than the regular example shown below:

```
        image[x][y] = vec3(R, G, B);
```

Step 3: Save the image to a file. To keep this workshop simple you will save the image to a file and use a photo editor to view it, rather than render the scene directly to the screen. Use the example code below to write your image to a file. You will also need to add `#include<fstream>` to use this function. Then open the file using a photo editor (e.g. Photoshop or GIMP) and check your image has the correct colour and dimensions.

```
// Save result to a PPM image
std::ofstream ofs("./untitled.ppm", std::ios::out | std::ios::binary);
ofs << "P6\n" << width << " " << height << "\n255\n";
for (unsigned y = 0; y < height; ++y)
{
        for (unsigned x = 0; x < width; ++x)
        {
              ofs << (unsigned char)(std::min((float)1, (float)image[x][y].x) * 255) <<
              (unsigned char)(std::min((float)1, (float)image[x][y].y) * 255) <<
              (unsigned char)(std::min((float)1, (float)image[x][y].z) * 255);
        }
}
ofs.close();
```

Part 2: Ray Casting

The next few steps are to implement the actual ray casting algorithm. The following is the pseudo-code for the process.

```
For each pixel:
        Construct a ray from camera through pixel
        Find intersections between the ray and primitives
        Compute colour of the pixel based on the nearest object
```

Step 4: Setup a loop to iterate through all of your pixels in your image. The next step is performed for each individual pixel in your image.

Step 5: Begin the ray casting algorithm by constructing a ray from camera through the current pixel. The following steps are to convert the pixel co-ordinates to the image plane:

- Normalise the pixel position to the range [0:1] using the screen dimensions:

$$PixelNormalized_x = \frac{(Pixel_x + 0.5)}{ImageWidth}$$

$$PixelNormalized_y = \frac{(Pixel_y + 0.5)}{ImageHeight}$$

  - **Note**: The small shift of +0.5 is so that the final ray passes through the centre of the pixel

- Remap the coordinates from the range [0:1] to [-1:1] and reverse the direction of the y-axis

$$PixelRemapped_x = 2 * PixelNormalized_x - 1$$

$$PixelRemapped_y = 1 - 2 * PixelNormalized_y$$

- So far this assumes the image is square to correct this the aspect ratio needs to be applied to the remapped coordinates. **Note**: as there are more pixels along the x-axis this only needs to be applied to the x-coordinate.

$$ImageAspectRatio = I_{ar} = \frac{ImageWidth}{ImageHeight}$$

$$PixelRemapped_x = (2 * PixelNormalized_x - 1) \ * I_{ar}$$

$$PixelRemapped_y = 1 - 2 * PixelNormalized_y$$

- Then incorporate the field of view (FOV) in terms of the angle α e.g. 30^o. You can multiply the screen pixel coordinates with the result of the tangent of this angle divided by two. **Note**: if you define the angle α in degrees you must convert it to radians.

$$PixelCamera_x = PixelRemapped_x * \tan\left(\frac{\alpha}{2}\right)$$

$$PixelCamera_y = PixelRemapped_y * tan\left(\frac{\alpha}{2}\right)$$

- As the point lies on the image plane which is 1 unit from the camera's origin you can express the final coordinate of the image on the image plane as:

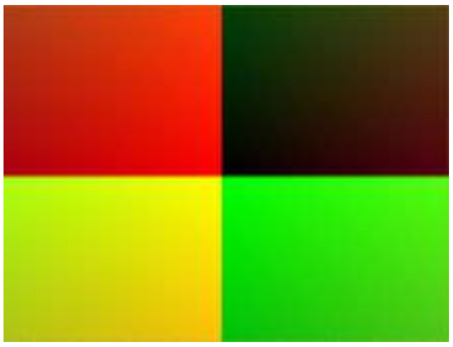
$$P_{cameraspace} = (PixelCamera_x, PixelCamera_y, -1)$$

Finally, you can compute a ray for this pixel by defining the origin of the ray as the camera's origin and the direction of the ray as a normalised vector. Initially, the camera origin and the world Cartesian coordinate system are the same when the camera is in its default position, therefore in this case the $rayOrigin$ is simply (0,0,0). The $rayOrigin$ will change if you add camera movement so it is best to leave it in the code even if it has no effect right now.

$$rayDirection \ = P_{cameraspace} - rayOrigin$$

  - **Hint**: It's a direction so don't forget to normalise the ray

To test if your camera rays were cast correctly, you can visualise the directions of the rays by setting each pixel in your image array to the colour or the $rayDirection$ e.g. r = x, g = y and z = b. If you run your code and open the generated image file you should now see the following:



If your result varies from this, examine the pattern you observe and consider what kind of error could produce it.