COMP 371: Computer Graphics
Final Project

by

Team 14

Concordia University
Department of Computer Science & Software Engineering (CSSE)

A Project Report Submitted Covering the project's software architecture, highlights of accomplishments, objectives, why it was interesting, Our objectives, what we learned, and the list of references and resources we have used.

Professor:
Serguei A. Mokhov, PhD

serguei.mokhov@concordia.ca

May, 2021

Team Members

#	Name	Student ID
1	Timmy Pierre Keo	40175104
2	Micheal Masciotra	40175211
3	Sarah Al Mamoun	40068208
4	Somar Jaafar	40152665
5	Jason Hilani	40066045

Date: **09/05/2021**

Acknowledgement

We would like to thank our TA Jonathan Llewellyn for his support throughout the semester. Without the labs, this project would have been impossible. We thank the school for giving us the grand opportunity to work as a team which has indeed promoted our team work spirit and communication skills. We also thank the individual group members for the good team spirit and solidarity.

Background

Throughout the semester, we became familiar with the basic techniques and concepts of 3D computer graphics. Applications include various sectors such as engineering, visualization, entertainment, gaming, etc.) The topics covered include 2D & 3D transformations, modeling and representation, illumination and shading, rendering, texturing, animation, physics-based animation, and the state-of-the-art software tools. We were taught the fundamental algorithms and techniques of how to program in modern OpenGL [5], a powerful software API used to produce high-quality computer-generated images of 2D & 3D scenes.

Use a package installer to install the following packages: `glfw, glew, glm`.

You will also need the following libraries:

- `OpenISS: https://github.com/OpenISS/OpenISS`
- `openFrameworks: https://openframeworks.cc/`

0.1 PA1: Modeling and Camera Control

We were given tasks which would allow us to get experienced with building simple virtual scenes consisting of various low poly meshes and objects, which mostly remain throughout the entire project development in PA2 and PA3. We needed to understand how to interact with the GPU for the first time in code using the OpenGL API, such as using vertex and fragment shaders. Each team member modelled letters and numbers from their names and student ID numbers, which were placed on a simple grid floor centered around a set of XYZ axis.

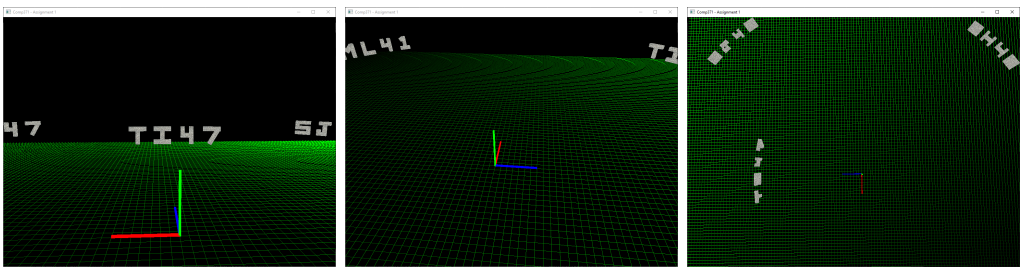


Figure 1: Screenshots from PA1

0.2 PA2: Lighting and Texture mapping

Starting from PA1, we extended the assignment and implemented shadows, lighting, and more modeling.

We had problems with trying to create OpenGL compatible objects using Blender[2]. We ended up having to triangulate the faces, and proceeded to UV-unwrap the model in order for it to be rendered correctly with the texture. This was used to create a half-circle 'stage' and we then placed our models from PA1 around it.

On our stage, we added a screen that presents a timed slideshow of screenshots of our models. This was simply done by changing the texture every few seconds. This screen became a placeholder for PA3.

We used the 'Phong illumination model' to implement a white point light source 30 units above the world's ground. For shadows, we used the two-pass shadow algorithm. The first pass renders the shadow from the point of view

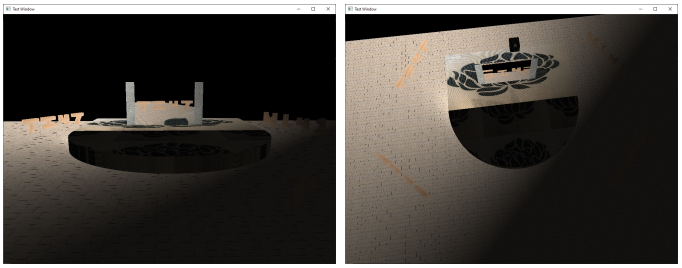


Figure 2: Screenshots from PA2

of our light, and the depth of each fragment is computed. Next, we render the scene normally but with an extra step to check if each fragment is in the shadow. This works perfectly for a directional light, since it simulates a sun and all the light rays point parallel in one direction. But, as we needed a point light, which bursts out light in all directions, the basic two pass algorithm is not sufficient, and we need omnidirectional shadow maps. To do this, we render the scene 6 times, for each direction of the point light's position, and generate a cubemap of the surrounding's depth. The cube map is used in the last render pass to check the depth of each fragment and display shadows.

Contents

Chapter 1

PA3 Introduction

1.1 Main Objectives

Our objective was to apply the skills we have accumulated over the span of the semester in computer graphics principles, as well as enhance our understanding of image processing and interactive programming skills. Our chosen topic, code name OIOFXCAM, was concerned with immersing a real time video feed from a webcam within our virtual graphics environment. This requires functionality for grabbing data frames from an external camera and converting each frame into an array of pixels which can be displayed as a moving texture. We needed to consult the openFrameworks method of implementation for webcams and translate it over to the OpenISS[1] framework.

Additionally, we will practice image processing by manipulating the raw camera feed into something more interesting, such as simulating a depth sensor and a Infrared sensor.

Chapter 2

Software Architecture

2.1 Interests

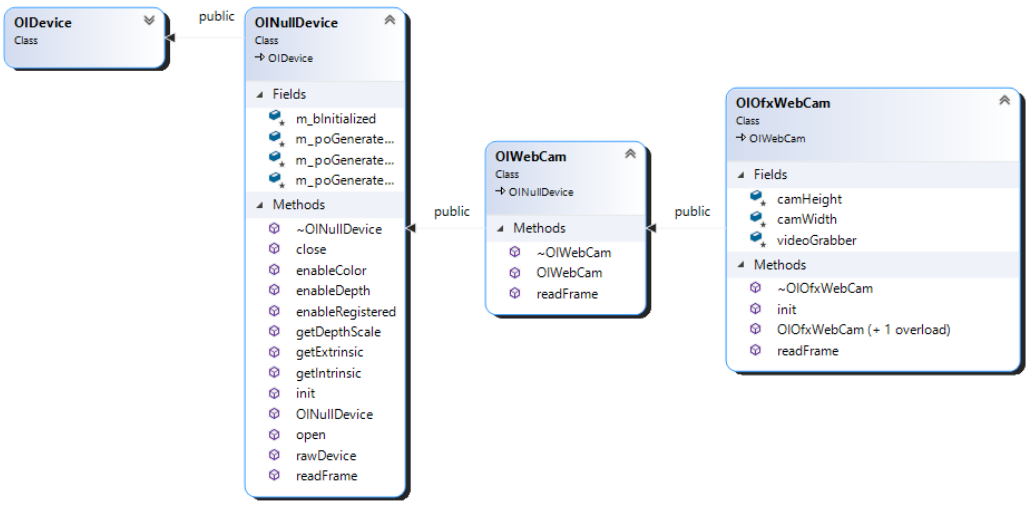
OIOFXCAM interested us initially because of its integration of the openFrameworks framework, which we viewed as a good introduction to creative coding principles. Specifically, the fact that openFrameworks runs on C++ and is built on top of OpenGL was exciting for us to apply the skills we learnt over the semester. In addition, the idea of grabbing real time video and manipulating it with 3D graphics introduces a range of great possibilities when it comes to computer vision and augmented reality (AR). Completing this project would be the first step to engaging in augmented reality applications for not only creative coding and augmented visual effects, but also the multitude of industry and educational use cases AR encompasses.

2.2 Architecture

We extended PA2 and included the OpenISS framework. We also extended the OpenISS framework to use openFrameworks[4]. We built an abstract Webcam class `OIWebCam` device that inherits from the `OINullDevice` in OpenISS, since `OINullDevice` already contains a default implementation for IR and Depth. Next, we inherited from our abstract class `OIWebCam` to create `OIOfxWebCam` which implements color stream based on the openFrameworks class for a webcam, `ofVideoGrabber`. The return value of the `readFrame()`

function creates an `OIDataFrame` object that holds information of the camera frame in OpenISS. `OIDataFrame` houses the texture buffer which we pass on to our OpenGL rendering pipeline to bind it as a texture.

In the `readFrame()` of `OIOfxWebCam`, we call the update method of the `ofVideoGrabber` object to get the current frame of the camera. That same object conveniently provides a `getPixels()` method, which gives an array of pixels (unsigned char) and can directly be passed as the data to an `OIDataFrame` object. In addition to `readFrame()`, the array of pixels are changed according to `StreamType`: `COLOR_STREAM` will simply return the array of pixels untouched; `DEPTH_STREAM` will convert all the colors of each pixel to grayscale using the ITU-R BT.709 standard for HDTV[3]; finally, `IR_STREAM` will switch the blue and green channels of each pixel to simulate an IR effect.



2.3 What we learned

We learnt the inner workings of the Device classes in OpenISS which makes it easier to handle any of the different types of devices possible within the framework, such as Kinect sensors and multiple webcams. We also got a much better understanding of texture buffers. In order to grab the image from openFrameworks and translate it over to the OpenISS, we had to take the array of pixels generated by the openFrameworks videoGrabber class and feed it into the OIDataFrame data array. We had to understand how a

video can be represented in code. We now understand that the video feed from the webcam is actually a series of images, one captured at each frame, played back at high speed to form motion. Each image is represented as a data array of chars, or simply 8-bit unsigned ints that represents the color. The positioning of the array also defines the color channel: for example, with RGB, representing the Red, Green, and Blue channels respectively, the index for R is 0, G is 1, and B is 2. This array is the basis of the OIDataFrame, and is what we use to bind to a texture placed in our world on the stage. We also now have a better understanding of image processing techniques which we had to learn in order to manipulate our video to simulate a depth sensor and an Infrared sensor. For the depth sensor look, we converted the image to Grayscale[3] using the formula below. The values of each color channel of a pixel is then replaced by that value.

$$Grayscale = 0.299R + 0.587G + 0.114B$$

To simulate Infrared, we simply swapped the Green and Blue channel values, effectively switching from RGB to RBG color space. These exercises helped us understand how to edit images by looping over each color channel and individual pixel for maximum creative control. We can use this concept to achieve all sorts of visual effects and aesthetics, not only for webcam feeds, but also textures in general.

Chapter 3

Challenges

3.1 Challenges Faced During Implementation

The main challenge we faced is linking the openFrameworks library to OpenISS with CMake. openFrameworks has lots of external libraries which had to be included as well, some of which did not work immediately resulting in an abundance of library rebuilds.

We were not aware of the many different libraries one can deal with and add to the project. There was also the issue of having to include directories. Previously, we did not have to deal with these issues. It was only OpenGL, GLEW, and GLFW that we dealt with.

Another challenge was having to adapt the relevant openFrameworks webcam device classes to the OpenISS OIDataFrame class. We had to figure out how to integrate these two frameworks and convert the data captured using the openFrameworks ofVideoGrabber class to our own OpenISS data buffer.

Chapter 4

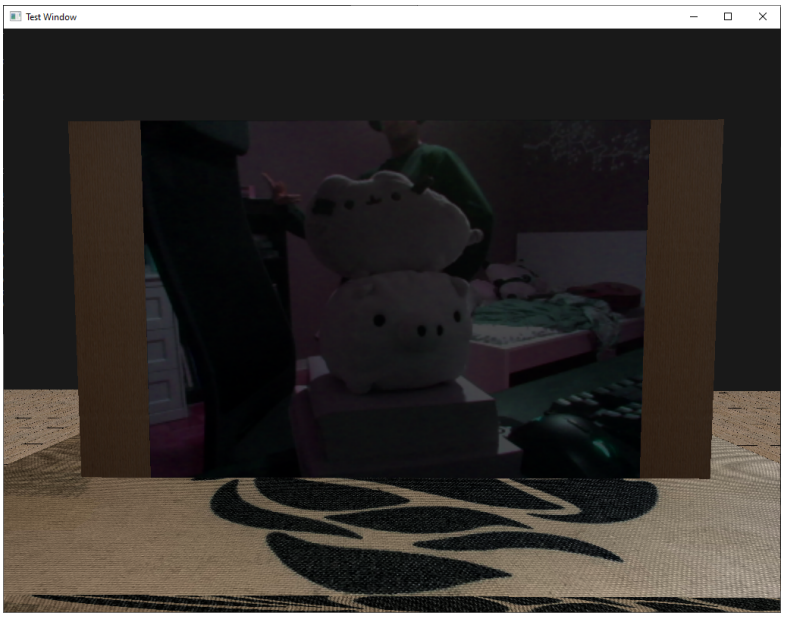
Demonstration

Here are screenshots that show the different implementations of color streams.

Key	What It does
WASD	Camera Movement (Up, Left, Down, Right)
8	Grayscale mode toggle
9	Infrared mode toggle

Color

Grayscale



IR

Chapter 5

Conclusion

Building up from PA1 where we learnt to render models and apply transformations to them, to PA2 where we added shadows and lighting, and finally to PA3 where we introduced OpenISS and openFrameworks – our final rendition of the project features the same letters from PA1 all sitting around a half-circle stage, with a live webcam feed playing. The live video can be switched between 3 modes: normal, Depth sensor, and Infrared.

The experience of linking multiple libraries and using them together to form a bigger project was a complex one. This was a particularly useful skill that we acquired while working on this as a team. OpenISS provides many possibilities as it abstracts multiple frameworks allowing them to more seamlessly work together to achieve great results. The project requirements were completed successfully on Windows. However, we were not able to build the libraries on Linux or macOS, which would be the first area of improvement. In addition, the depth sensor and Infrared are not accurate but merely simulated, so implementing the real version of each would be an interesting additional challenge. Otherwise, an endless number of visual effects and creative coding techniques could be applied on the video for more fun experiments.

References

[1] Serguei A. Mokhov et al. *OpenISS Open Illimitable Space System*. 2016-2021. URL: https://github.com/OpenISS/OpenISS.

[2] *Blender*. URL: https://www.blender.org/.

[3] *Grayscale*. URL: https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems (visited on 2021).

[4] *openFrameworks*. URL: https://openframeworks.cc/ (visited on 2021).

[5] Joey de Vries. *LearnOpenGL*. URL: https://www.learnopengl.com/.