# CSE 165 Final Project Report

Ahmad Suleiman

Jophiel Nino

Angel Jaimez

CSE 165

Intro to Object Oriented Programming

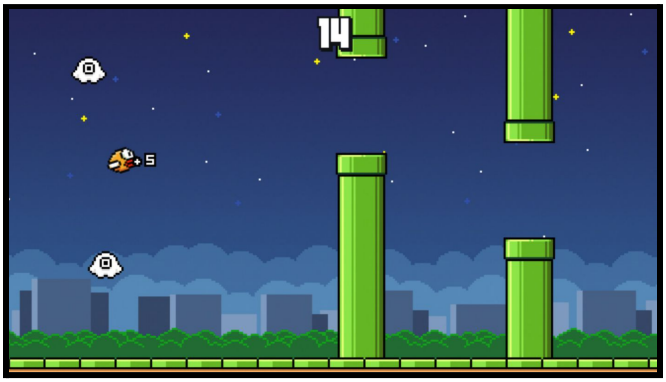
University of California, Merced

Spring 2021

# Motivation

For our final project in the class CSE 165 (Object Oriented Programming) at UC Merced, we were designated to create an interactive game using OOP principles learned throughout the semester. In order to do this, we were encouraged to develop an OpenGL game based on C++ that utilizes several libraries such as SOIL2, Glut, Glew, and GLFW. Through the use of all the Library's available to us we were able to create a Flappy Bird remake of the classic game.

## *Chosen Game - Flappy Bird*




We decided to build our project revolving around the popular game known as Flappy Bird, a simple 2 dimensional side-scrolling game. The objective of this game is simple: tap the screen to have the bird flap its wings as it traverses through an infinite series of randomly generated pipes. Score is measured by every pipe the bird passes through, and continues infinitely until the bird collides with one of the pipes. We chose this game in particular because we all had a history with this specific game, and remember it as a simple yet attention grabbing game notorious for being unexplainably difficult. Another reason we added this game was because of how simple this game was, yet it had a grip on the people who played it as it was simple but difficult to play to last as long as you could. Because of this, the simple game of Flappy Bird drew the people who played it back to their phones to play it some more.

# Methodology

To create our two dimensional platform through C++, we were designated to use several libraries supported across the various platforms such as Windows, MacOS, and Linux. These libraries and frameworks include OpenGL, Glut, GLFW, SOIL, and Object Oriented Programming.

## *OpenGL*

OpenGL is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. It supports various languages and platforms but in our case, we initiate it with C++ programming. In the development of our game, most of the backend work and rendering is done by OpenGL, but shows very little presence in our code as we connect to it through the use of other libraries.

## *GLFW*

GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, receiving input and events, and makes up most of the programming in our project. While OpenGL is the rendering agent that outputs the scene of our game, GLFW is the logical factor that allows us to implement all the game mechanics, including but not limited to window creation, input mapping and checking, time tracking, and OpenGL initialization. We consider it to be more integral to our game than any other library added, and its simplicity in design made this project's completion far easier to achieve.

### *SOIL2*

Soil2, an update fork from the original Soil library, is a small library that simplifies the process of uploading textures into OpenGL shaders. It overrides the process of using stb_image, the default texture uploading method of OpenGL, and allows users to use a much simpler method of uploading through Soil. In the development of our game, we found it incredibly simple in implementation and very useful in shortening our overall code, reducing the process of 20+ lines for every upload into a mere ~4 lines of code. These mere few lines then allowed us to display our own images we downloaded from the internet to be used as our sprites for our game which include the bird, pipes, background, and floor. This helped to make the game look like a smooth clean game with nice sprites.

### *Object Oriented Programming*

Object Oriented Programming is an extremely common practice in Computer Science and Programming, based on the concept of code and memory representing ‘objects’, storages of different levels of data and more code known as procedures. In the C++ language, the ability to create object oriented code is the biggest factor that distinguishes the language from its predecessor, C. Through the use of Object Oriented Programming we were able to combine all our knowledge as well as help from the internet for references, we were able to create a remake of the classic phone game Flappy Bird.

## Implementation

The development of the game itself was a difficult experience as none of us had any previous experience with working with the required libraries such as Soil2, GLFW, and OpenGL. We

were required to follow multiple guides online on how to implement OpenGL into a given project and how to initialize an OpenGL/Make friendly environment. The first process in developing our game focused heavily on establishing a working environment within the Visual Studio IDE. This required creating the Visual Studio project folders, downloading our libraries sources, and compiling them on various machines to achieve support for multiple platforms. When this was completed, we had a fully working directory, preloaded with all the libraries and their headers such as GLFW and SOIL, as well as the Visual Studio Solution file which was capable of opening and running our program. After all the libraries were properly linked and included in the build of our program, we were finally able to start the development of our game.

### *Objects*

The concept of Flappy Bird is simple, a bird flying through a series of pipes in a fairly static environment. Knowing this, we understood that our project would have to object-ify all of these elements so we created header files for the bird, pipes, and all environmental features. Each of these header files contain standard functions found in all the objects, such as texture loading (using SOIL), reinitialization for when coordinates change, and movement functions (if needed). Another important decision made in the design of our objects was omitting the use of cpp files to initialize declared member functions of these headers/objects. We found various issues in the compilation process with these files, so we simply include all our initializations in the header files themselves.

### *Scene Design*

Designing the scene and rendering the game came easy to us, as every object in the scene had their dedicated initialization functions. It was as easy as initializing instances of each object,

passing appropriate coordinate data for placement, and passing them into a draw function that would run in our event loop, depending on the needed scene. We have a total of three scenes that render depending on the status of the game:

1. The Start Screen, which displays upon entering the program. It shows all our little sprites as well as a game logo and start button
2. The Game Screen, which displays when the game has started. It invokes all the game mechanics such as collision, speed, and movement.
3. The Death Screen, which displays a simple 'You Died' screen, inspired by the game Dark Souls. It only renders one element.

The rendering of all our objects relied heavily on the **shaderClass**, a class we downloaded from the LearnOpenGL guide that is required for OpenGL use. It declares how our shaders are generated and what data is expected when we pass our instructions to the shader creation functions within the header files. These instructions exist in separate files known as *default.vert* and *default.frag,* which initializes shaders based on the vertice placement passed for each product.

## *Movement*

Movement was also fairly easy to implement, where our program utilizes the simple input tracker provided by GLFW and changes the coordinates of any object specified accordingly. When it comes to moving our scene, we found it most optimal to just change the horizontal movement of the pipes rather than the bird, and infinitely recycle between our initialized pipes to better conserve memory. So with the pipes moving towards the bird at an increasing speed, we used input mapping to change the bird's vertical placement so that the user can avoid the incoming pipes, and play the game in that manner.

### *Collision Detection*

With all the movement and scene rendering in place, the last element required to make this a challenging game was to add collision detection, such that whenever the main bird sprite comes into contact with any of the pipes, the game promptly ends. To do this, we implemented a collision detection function that is constantly called in the event loop, which tracks the coordinates of both the bird and the pipes. This was fairly easy because of our use of object oriented programming on these elements. Using several if statements, the function checks to see if any of the birds corners are in the space of any of the pipes, and if so, calls our game end function which abrupts movement and tracking and changes the scene to our dedicated death screen.

## Works Cited


https://www.glfw.org/

https://en.wikipedia.org/wiki/OpenGL

https://github.com/SpartanJ/SOIL2

https://en.wikipedia.org/wiki/Object-oriented_programming

https://learnopengl.com/Getting-started/Shaders