

**Implementation of Object file loader using C**

***Project Report***

*Submitted in partial fulfillment of the requirements for the award of the degreeof*

## BACHELOR OFTECHNOLOGY

**in**

**COMPUTER SCIENCE & ENGINEERING**

|  |  |  |
| --- | --- | --- |
| Ishika Agrawal | R-164218097 | CSEIOT |
| Silvi | R-164218072 | CSE IOT |
| Shreya Sharma | R-164218070 | CSE IOT |
| Manya Mishra | R-103218080 | CSE BAO |

***Under the guidance of***

**Mr Mrinal Goswami**

Assistant Professor (SS)



**School of Computer Science Department of Systematics**

**UNIVERSITY OF PETROLEUM AND ENERGY STUDIES**

**Bidholi, Via Prem Nagar, Dehradun, Uttarakhand**



## CANDIDATE’S DECLARATION

We hereby certify that the project work entitled **“Implementation of Object file loader using C”**in partial fulfillment the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND

ENGINEERING with specialization in “**Internet of Things & Smart Cities and ”** and submitted to the School of Computer Science, Department of Systemics and Department of Informatics, University of Petroleum & Energy Studies, Dehradun, is an authentic record ofour work carried out during a period from **Aug**-**2020** to **DEC-2020** under the supervision of **Dr. Mrinal Goswami, Assistant Professor, Dept. of Systemics.**

This is to certify that the above statement made by the candidate is correct to the best of myknowledge.

.

Date:12 December 2020

**Dr. Neelu Jyoti Ahuja Dr. Mrinal Goswami**

HOD – Systemics Project Guide

School of Computer Science

University of Petroleum & Energy Studies Dehradun – 248 001 (Uttarakhand)

# ACKNOWLEDGEMENT

We wish to express our deep gratitude to our guide **Dr. Mrinal Goswami** , for all advice, encouragement and constant support she has given us throughout our project work. This work would not have been possible without her support and valuable suggestions.

We sincerely thank to our respected Program Head of theDepartment,

**Dr. Neelu Jyoti Ahuja**, for her great support in doing our project.

We would like to thank all our **friends** for their help and constructive criticism during our project work. Finally, we have no words to express our sincere gratitude to our **parents** who have shown us this world and for every support they have given us.

**ABSTRACT**

OBJ is a geometry definition file format first developed by [Wave frontTechnologies](https://en.wikipedia.org/wiki/Wavefront_Technologies) for its [Advanced Visualizer](https://en.wikipedia.org/wiki/The_Advanced_Visualizer) animation package. The file format is open and has been adopted by other 3D graphics application vendors.

The OBJ file format is a simple data-format that represents 3D geometry alone — namely, the position of each [vertex](https://en.wikipedia.org/wiki/Vertex_(geometry)), the [UV position](https://en.wikipedia.org/wiki/UV_mapping) of each texture coordinate vertex, [vertex normal](https://en.wikipedia.org/wiki/Vertex_normal), and the faces that make each polygon defined as a list of vertices, and texture vertices. Vertices are stored in a counter-clockwise order by default, making explicit declaration of face normal unnecessary. OBJ coordinates have no units, but OBJ files can contain scale information in a human readable comment line.

This is a basic .obj loader written in C. A C wrapper is included. It can parse vertices, texture coordinates, normal, 3 or 4 vertex faces, and .obj files. The OBJ file is in readable format you can figure out the contents of any of the files simply by opening them in OpenGL. The contents should reveal an easy structure for materials. When an object loader's activation function is called, it should open the object file and try to recognize its contents. Each object loader is therefore responsible for identifying the files it can load. On the other hand, the loader understands the object file, it reads the file and submits its contents to the host.

There is also support for non-standard object types that are relevant to raytracing.When you need to draw a more advanced object like character, house, terrain, vehicles we can't pass the vertices by ourselves for these objects so we have to use the 3D model and model are simply the meshes made of one or more number of vertices. There are many 3D modelling software’s one of them is OpenGL which is open source also.

This project is a simple obj file loader that loads the vertex, normal and texture data from an .obj file and arranges in an array to be loaded into OpenGL programs.

# TABLE OF CONTENTS

1. Introduction
2. Problem Statement
3. Motivation
4. Objective
5. Literature Review
6. Methodology
7. Design and Implementation
   * 1. A Basic Object File
     2. Implementation
     3. A quick list of what the parser can handle
     4. Pseudo Code
     5. Proposed Technique (Algorithm) & Implementation
     6. Reading and Parsing Stage
     7. Code
     8. Snapshot and Screenshot
8. Plan of Work
9. Features
10. Future Enhancements
11. References

# INTRODUCTION

A simple obj file loader that loads the vertex, normal and texture data from an .obj file and arranges in an array to be loaded into OpenGL programs.

Object file format contains a 3-dimensional object that contains a range of information, from coordinates and textures, to posture and pose.

The OBJ file format is a simple data-format that represents 3D geometry alone — namely, the position of each [vertex](https://en.wikipedia.org/wiki/Vertex_(geometry)), the [UV position](https://en.wikipedia.org/wiki/UV_mapping) of each texture coordinate vertex, [vertexnormal](https://en.wikipedia.org/wiki/Vertex_normal), and the faces that make each polygon defined as a list of vertices, and texture vertices. Vertices are stored in a counter-clockwise order by default, making explicit declaration of face normal unnecessary. OBJ coordinates have no units, but OBJ files can contain scale information in a human readable comment line.

There is also support for non-standard object types that are relevant to raytracing. When you need to draw a more advanced object like character, house, terrain, vehicles we can't pass the vertices by ourselves for these objects so we have to use the 3D model and model are simply the meshes made of one or more number of vertices.

OBJ is a geometry definition file format first developed by [Wave front Technologies](https://en.wikipedia.org/wiki/Wavefront_Technologies) for its [Advanced Visualizer](https://en.wikipedia.org/wiki/The_Advanced_Visualizer) animation package. The file format is open and has been adopted by other 3D graphics application vendors.

# PROBLEM STATEMENT

Object file format contains a 3-dimensional object that contains a range of information, from coordinates and textures, to posture and pose.

The OBJ file format is a common format and, as a result, needs to be exported into a number of 3D software applications for viewing and editing purposes. Therefore, there is a need to develop a file loader that exports the object file to one such tool for opening Object files.

# MOTIVATION

Object file format contains a 3-dimensional object that contains a range of information, from coordinates and textures, to posture and pose.

The OBJ file format is a common format and, as a result, needs to be exported into a number of 3D software applications for viewing and editing purposes. Therefore, there is a need to develop a file loader that exports the object file to one such tool for opening Object files.

# OBJECTIVE

The objective of this project is to develop a tool which could export an object file and open it in a 3D software for viewing and editing purposes.

 The sub-objectives of the project are:

1. To export the object file to a 3D software application like OpenGL.

2. To view and edit object files.

# LITERATURE REVIEW

1. According to Paul Bourke the .obj file format supports both polygonal objects and free-form objects. Polygonal geometry uses points, lines, and faces to define objects while free-form geometry uses curves and surfaces.[1]
2. According to Wave front Technologies the OBJ file format supports lines, polygons, and free-form curves and surfaces. Lines and polygons are described in terms of their points, while curves and surfaces are defined with control points and other information depending on the type of curve. [2]

# System Requirements

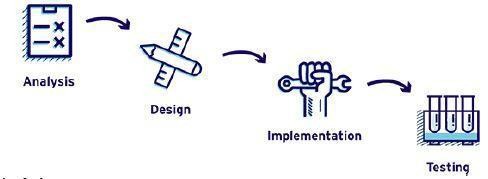
**Software Requirements:** OpenGL, Code::Blocks, Blender, Visual Studio

**Hardware Requirements:** System with 64-bit processor

Windows 10

4 GB RAM

# METHODOLOGY



*Fig. 1 : Waterfall Model*

The project is divided into four major section:

* Requirement Analysis

Studying the existing techniques and journals published on the .obj file loader

Assessment of the requirements of the project

Defining the scope of the project

* Design and development of prerequisites

Defining the various aspects of the project

Preparing the flow diagram of the project

Preparing the various modules of the project .

* Implementation & unit testing

Creating the modules using C language.

Integrating the modules

Functioning of different transitions of probabilities in model

Testing each module separately.

* Testing

Testing the integration of modules.

Testing the system.

# DESIGN AND IMPLEMENTATION

1. **A Basic object file**

An OBJ file looks more or less like this:

v 1.000000 -1.000000 -1.000000

v 1.000000 -1.000000 1.000000

v -1.000000 -1.000000 1.000000

v -1.000000 -1.000000 -1.000000

vt 0.748573 0.750412

vt 0.749279 0.501284

vt 0.999110 0.501077

vt 0.999455 0.750380

vt 0.250471 0.500702

vt 0.249682 0.749677

vt 0.498993 0.250415

vt 0.748953 0.250920

vn 0.000000 0.000000 -1.000000

vn -1.000000 -0.000000 -0.000000

vn 0.000000 1.000000 -0.000000

vn -0.000000 -1.000000 0.000000

f 5/1/1 ½/1 4/3/1

f 5/1/1 4/3/1 8/4/1

f 5/1/6 6/10/6 2/9/6

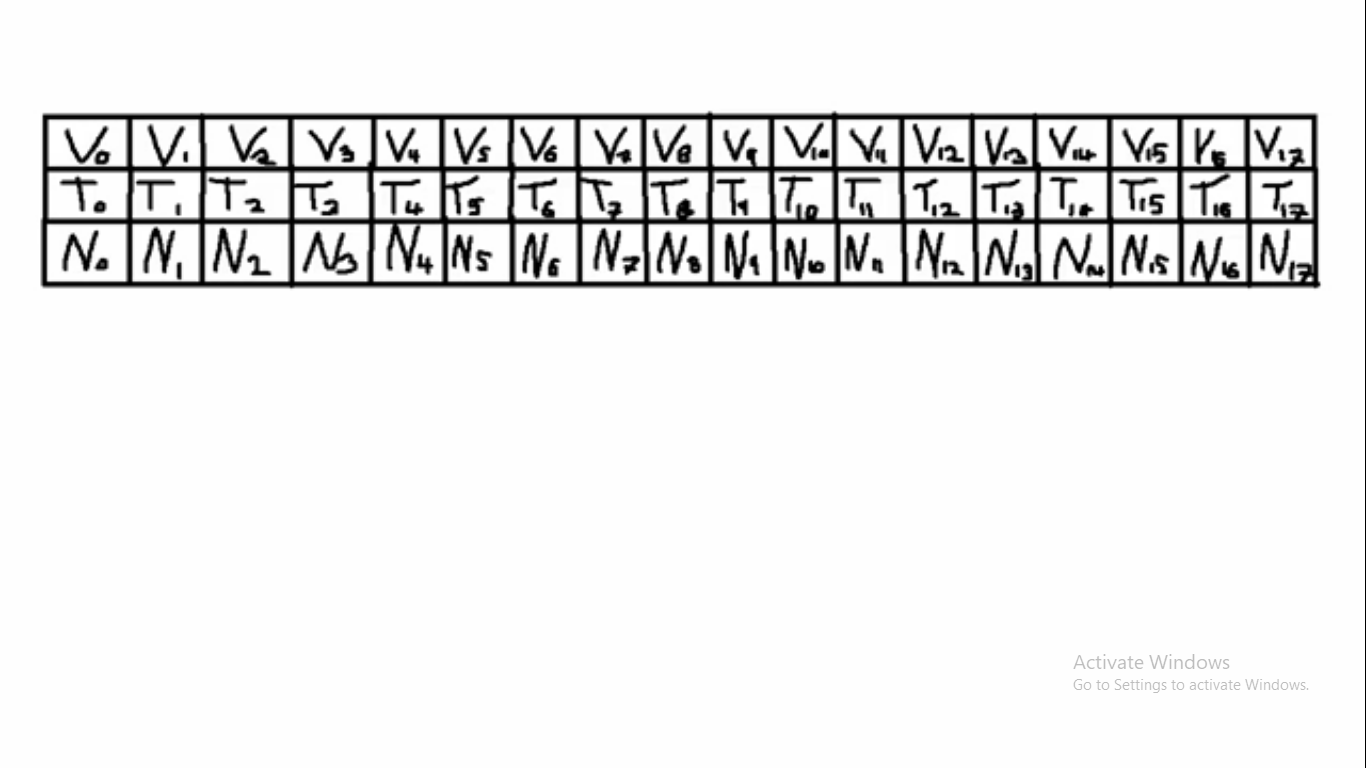
f 5/1/7 8/11/7 6/10/7

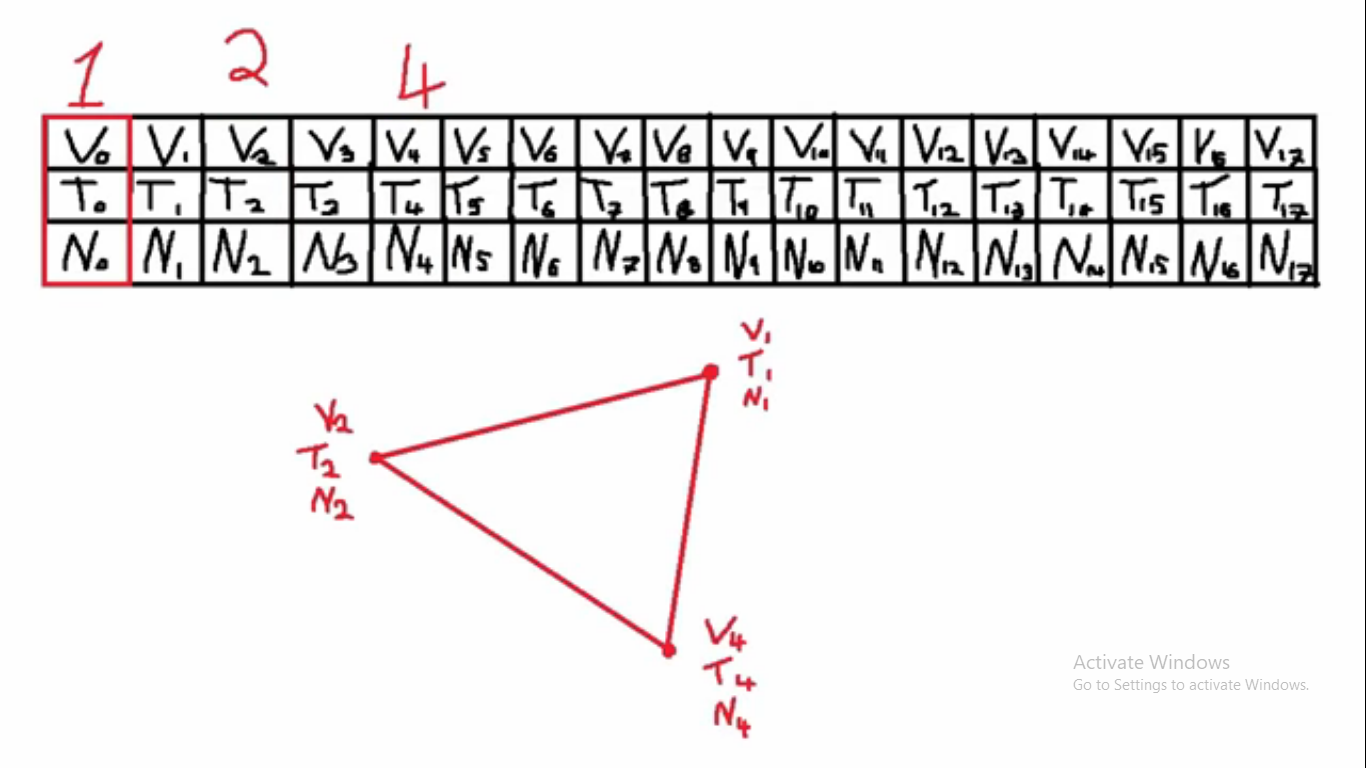
f 8/11/7 7/12/7 6/10/7

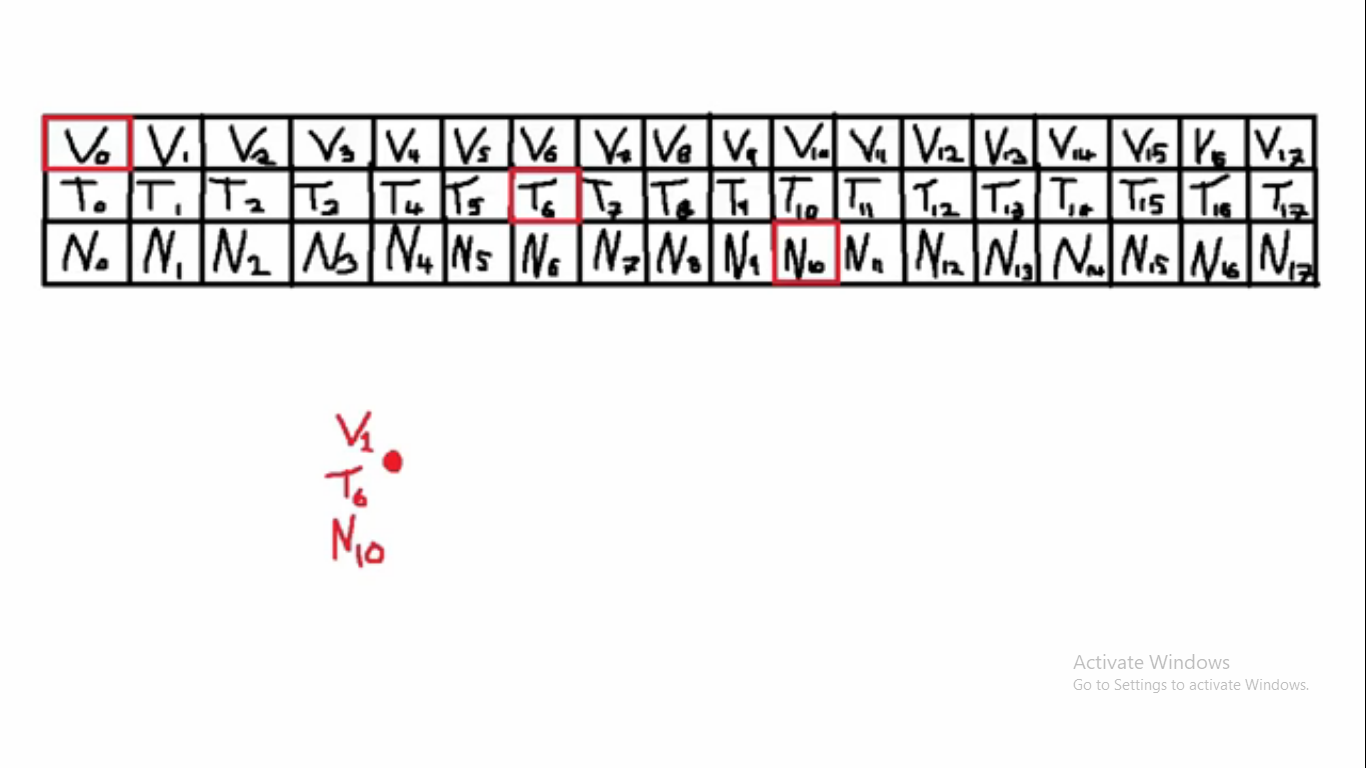
f ½/8 2/9/8 3/13/8

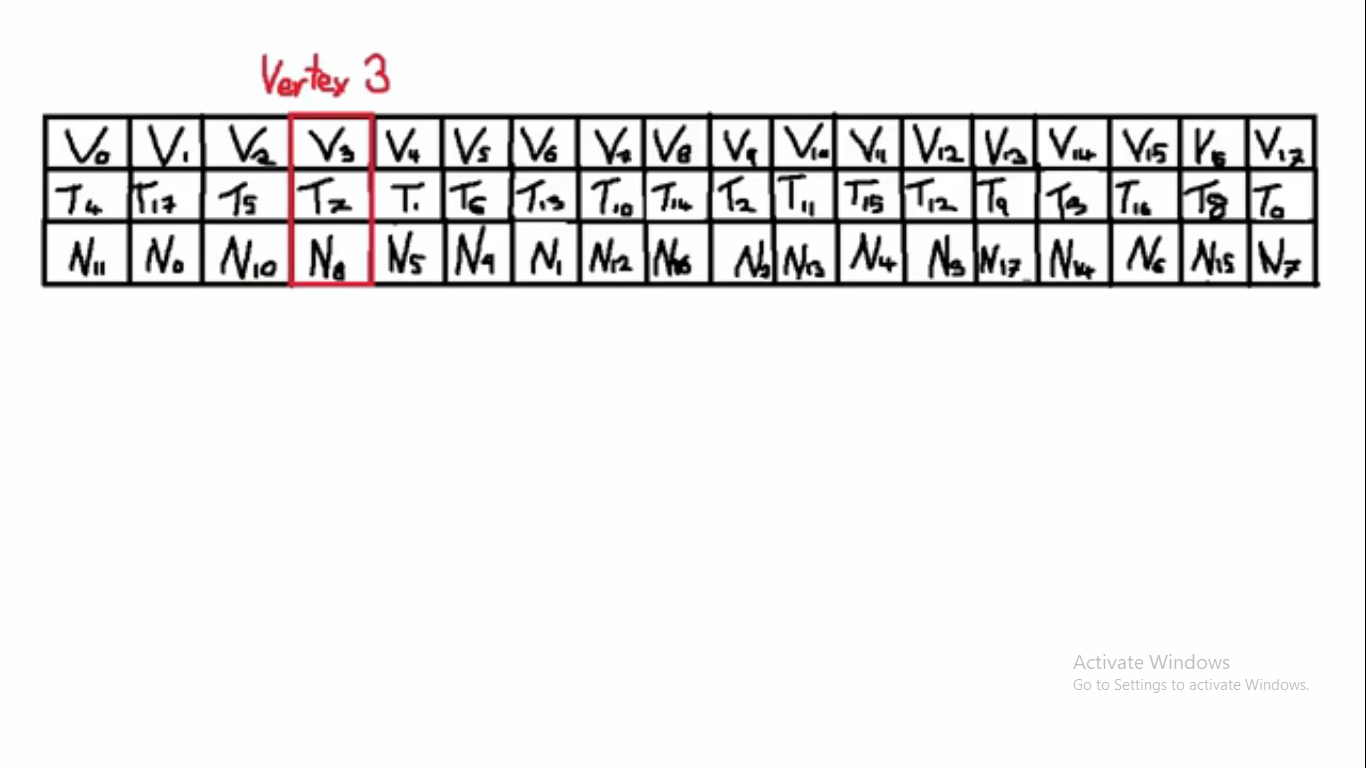
f ½/8 3/13/8 4/14/8

1. **Implementation**





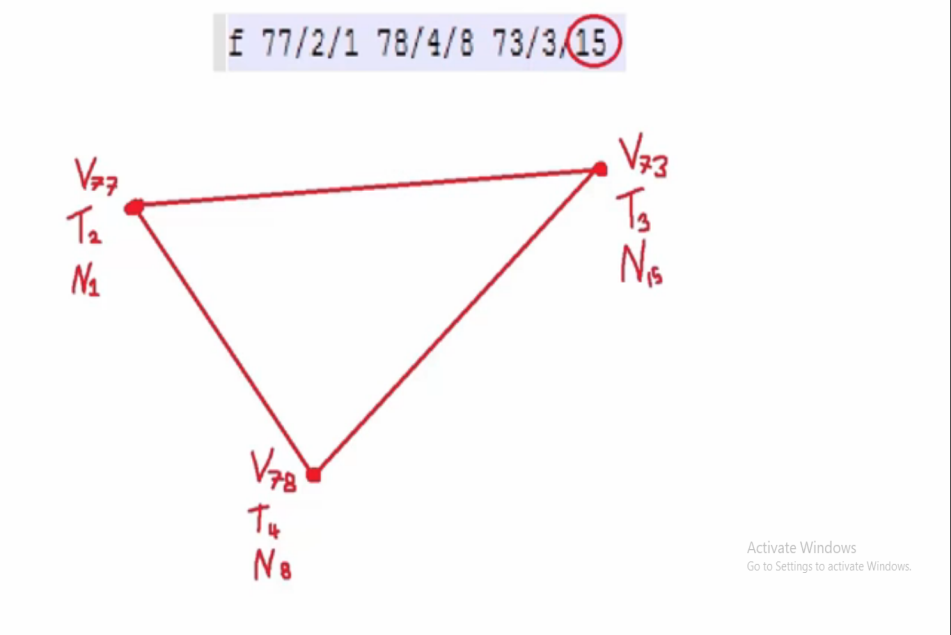




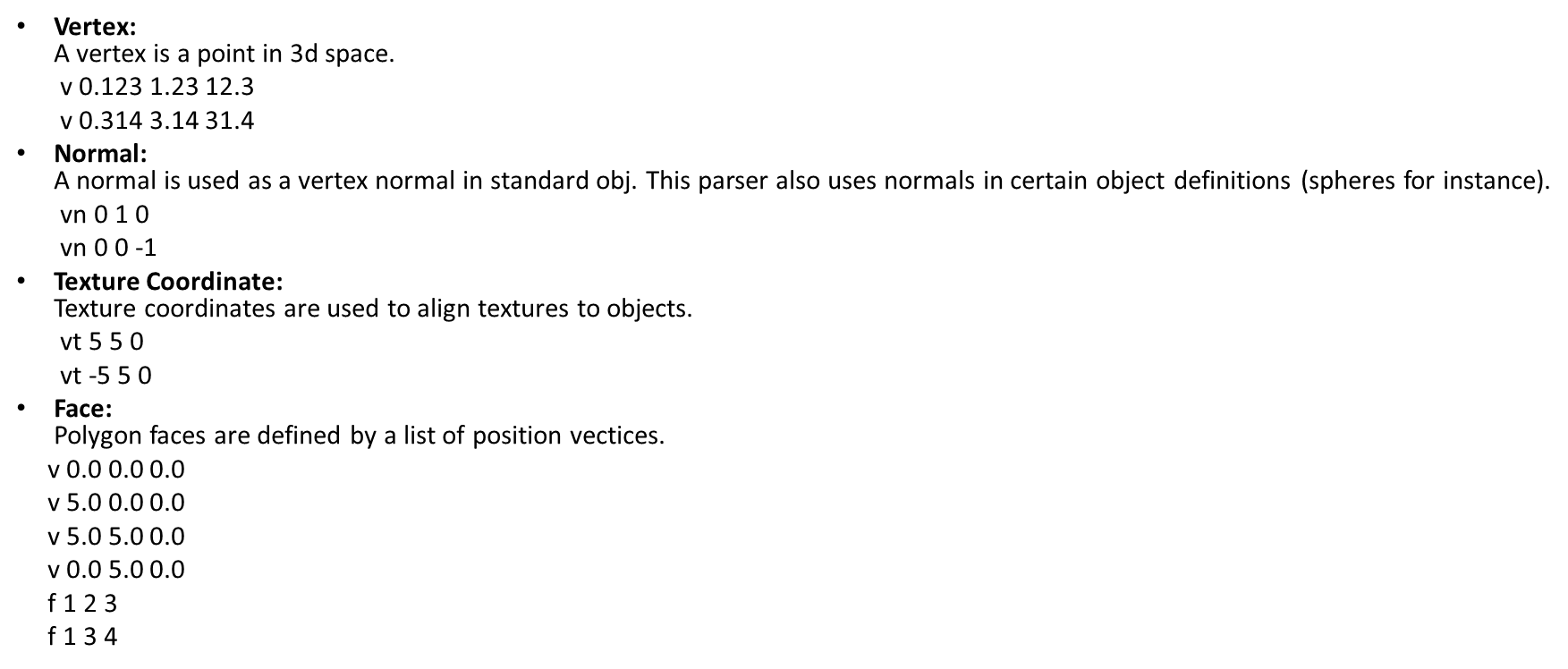
So, for f 77/2/1 78/4/8 73/3/15 :

* 77/2/1 describes the first vertex of the triangle
* 78/4/8 describes the second vertex of the triangle
* 73/3/15 describes the third vertex of the triangle
* For the first vertex, 77 says which vertex to use.
* 2 says which texture coordinate to use.
* 1 says which normal to use.

These numbers are called indices. It’s handy because if several vertices share the same position, you just have to write one “v” in the file, and use it several times. This saves memory.



1. **A quick list of what the parser can handle**



1. **Pseudo Code**
2. Read the data from the .obj file of the given 3D object.
3. Check if the data is for vertexPosition
4. Create new PositionNode and fill data
5. Adding Position Node to the end of the Linked List
6. Create new TextureNode and fill data
7. Add Texture Node to end of Linked List
8. Create new TextureNode and fill data
9. Add Texture Node to end of Linked List
10. Move the file pointer 100 characters back.
11. Allocate arrays for temporary storage of attribute data.
12. Populate arrays from respective linked lists.
13. Make sure that the object has textures applied.
14. Use face indices to build finalBuffer.
15. Load the object dimensions.
16. **Proposed Technique (Algorithm) & Implementation**

* Input: OBJ file.
* Output: V vector of 3D coordinates,

VT vector of 2D coordinates,

VN vector of normal vectors,

F vector of indices.

/\*CPU OBJ Read\*/

* for each line in OBJ file do
* if line == v then

Parse line as vertex

* append vertex to tempP
* else if line == vt then

Parse line as vt

* append vt to tempT
* else if line == vn then

Parse line as normal

* append vt to tempN
* else if line == f then

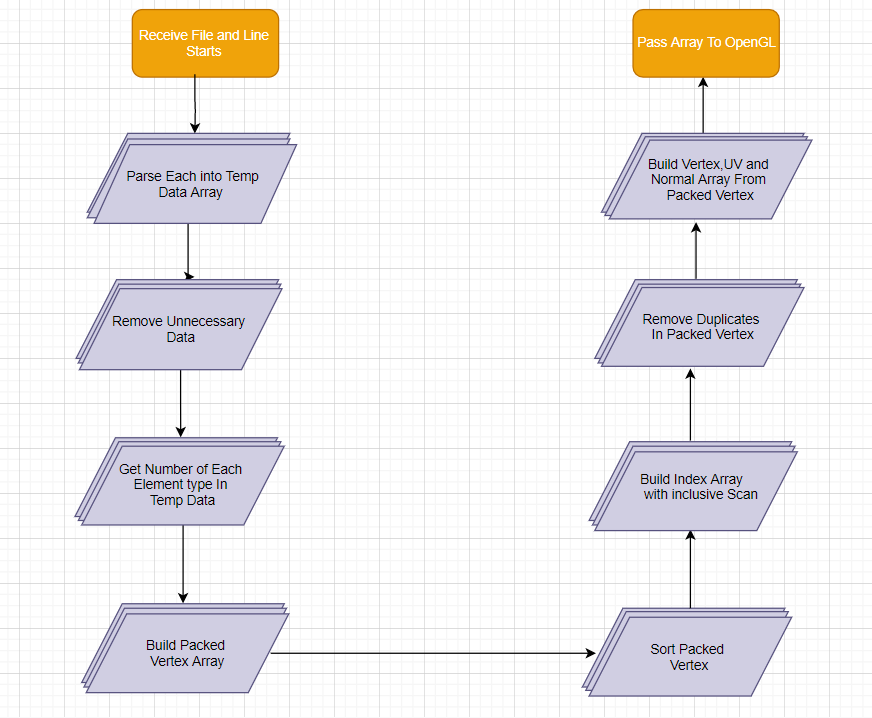
Parse line as face

* Use face indices to build finalBuffer from tempP,

tempT and tempN

* end if
* end for
* return finalBuffer

1. **Reading and Parsing stage**



1. **Code**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include<ctime>

#include <stdlib.h>

#include <string.h>

#include <iostream>

#define DEBUG\_INFO

typedef struct

{

float x;

float y;

} vec2;

typedef struct

{

float x;

float y;

float z;

} vec3;

static int num\_posNodes = 0;

static int num\_texNodes = 0;

static int num\_norNodes = 0;

static int num\_indexNodes = 0;

typedef struct PositionNode {

vec3 pos;

struct PositionNode\* nextNode;

} pNode;

typedef struct TextureNode {

vec2 pos;

struct TextureNode\* nextNode;

} tNode;

typedef struct NormalNode {

vec3 pos;

struct NormalNode\* nextNode;

} nNode;

typedef struct IndexNode {

int p1, p2, p3;

struct IndexNode\* nextNode;

} iNode;

//Main loading function

float myobj\_load(const char\* fileName, float\* faces) {

FILE\* file = fopen("stall.obj", "r");

char dataType[20];

float\* positionArray = NULL;

float\* textureArray = NULL;

float\* normalArray = NULL;

bool textureAvailable = false;

int f\_count = 0;

char readVal[30];

float\* finalBuffer = NULL;

if (file == NULL) {

printf("Unable to open File");

return 0;

}

pNode\* headP = NULL;

pNode\* currentNodeP = NULL;

pNode\* tempP = NULL;

tNode\* headT = NULL;

tNode\* currentNodeT = NULL;

tNode\* tempT = NULL;

nNode\* headN = NULL;

nNode\* currentNodeN = NULL;

nNode\* tempN = NULL;

iNode\* headI = NULL;

iNode\* currentNodeI = NULL;

iNode\* tempI = NULL;

while (1) {

int result = fscanf(file, "%s", &dataType);

if (result == EOF)

break;

if (strcmp(dataType, "v") == 0) { //checking if the data is for vertexPosition

float val1, val2, val3;

fscanf(file, "%f %f %f", &val1, &val2, &val3);

tempP = (pNode\*)malloc(sizeof(pNode)); //creating new PositionNode and filling data

tempP->pos.x = val1;

tempP->pos.y = val2;

tempP->pos.z = val3;

if (headP != NULL) { //adding Position Node to end of Linked List

currentNodeP->nextNode = tempP;

currentNodeP = tempP;

currentNodeP->nextNode = NULL;

}

else {

headP = currentNodeP = tempP;

}

++num\_posNodes;

}

else if (strcmp(dataType, "vt") == 0) {

float val1, val2;

fscanf(file, "%f %f", &val1, &val2);

tempT = (tNode\*)malloc(sizeof(tNode)); //creating new TextureNode and filling data

tempT->pos.x = val1;

tempT->pos.y = val2;

if (headT != NULL) { //adding Texture Node to end of Linked List

currentNodeT->nextNode = tempT;

currentNodeT = tempT;

currentNodeT->nextNode = NULL;

}

else {

headT = currentNodeT = tempT;

}

++num\_texNodes;

}

else if (strcmp(dataType, "vn") == 0) {

float val1, val2, val3;

fscanf(file, "%f %f %f", &val1, &val2, &val3);

tempN = (nNode\*)malloc(sizeof(nNode)); //creating new TextureNode and filling data

tempN->pos.x = val1;

tempN->pos.y = val2;

tempN->pos.z = val3;

if (headN != NULL) { //adding Texture Node to end of Linked List

currentNodeN->nextNode = tempN;

currentNodeN = tempN;

currentNodeN->nextNode = NULL;

}

else {

headN = currentNodeN = tempN;

}

++num\_norNodes;

}

else if (strcmp(dataType, "f") == 0) {

//move the file pointer 100 characters back. \*\*Note: I am assuming that no line will be longer than that\*\*

fseek(file, -100, SEEK\_CUR);

//allocating arrays for temporary storage of attriute data

#ifdef DEBUG\_INFO

printf("Positions: %d\nTextures: %d\nNormals: %d\n", num\_posNodes, num\_texNodes, num\_norNodes);

#endif

positionArray = (float\*)malloc(sizeof(float) \* 3 \* num\_posNodes);

if (headT != currentNodeT)

textureArray = (float\*)malloc(sizeof(float) \* 2 \* num\_texNodes);

normalArray = (float\*)malloc(sizeof(float) \* 3 \* num\_norNodes);

//populating arrays from respective linked lists

int i = 0;

for (pNode\* reader = headP; reader != currentNodeP->nextNode; reader = reader->nextNode) {

positionArray[i] = reader->pos.x;

positionArray[i + 1] = reader->pos.y;

positionArray[i + 2] = reader->pos.z;

#ifdef DEBUG\_INFO

printf("Vertex %d: %f %f %f\n", i / 3, positionArray[i], positionArray[i + 1], positionArray[i + 2]);

#endif

i += 3;

}

// making sure that the object has textures applied

if (headT != currentNodeT) {

i = 0;

for (tNode\* reader = headT; reader->nextNode != NULL; reader = reader->nextNode) {

textureArray[i] = reader->pos.x;

textureArray[i + 1] = reader->pos.y;

#ifdef DEBUG\_INFO

printf("Texture %d: %f %f\n", i / 2, textureArray[i], textureArray[i + 1]);

#endif

i += 2;

}

textureAvailable = true;

}

i = 0;

for (nNode\* reader = headN; currentNodeN->nextNode != reader; reader = reader->nextNode) {

normalArray[i] = reader->pos.x;

normalArray[i + 1] = reader->pos.y;

normalArray[i + 2] = reader->pos.z;

#ifdef DEBUG\_INFO

printf("Normal %d: %f %f %f\n", i / 3, normalArray[i], normalArray[i + 1], normalArray[i + 2]);

#endif

i += 3;

}

while (fgets(readVal, 30, file)) {

if (readVal[0] == 'f')

++f\_count;

#ifdef DEBUG\_INFO

printf("Face: %d\n", f\_count);

#endif

}

fseek(file, 0, SEEK\_SET);

break;

}

}

// TODO now deal with the 'f' section of the file, this is going to be the real work

if (!textureAvailable) {

finalBuffer = (float\*)malloc(sizeof(float) \* f\_count \* 6 \* 3);

f\_count = 0;

while (1) {

int result = fscanf(file, "%s ", &dataType);

if (result == EOF)

break;

int positions[3];

int normals[3];

if (strcmp(dataType, "f") == 0) {

fscanf(file, "%d//%d %d//%d %d//%d", &positions[0], &normals[0], &positions[1], &normals[1], &positions[2], &normals[2]);

#ifdef DEBUG\_INFO

printf("%d//%d %d//%d %d//%d\n", positions[0], normals[0], positions[1], normals[1], positions[2], normals[2]);

#endif

finalBuffer[f\_count] = positionArray[3 \* (positions[0] - 1)];

finalBuffer[f\_count + 1] = positionArray[3 \* (positions[0] - 1) + 1];

finalBuffer[f\_count + 2] = positionArray[3 \* (positions[0] - 1) + 2];

finalBuffer[f\_count + 3] = normalArray[3 \* (normals[0] - 1)];

finalBuffer[f\_count + 4] = normalArray[3 \* (normals[0] - 1) + 1];

finalBuffer[f\_count + 5] = normalArray[3 \* (normals[0] - 1) + 2];

finalBuffer[f\_count + 6] = positionArray[3 \* (positions[1] - 1)];

finalBuffer[f\_count + 7] = positionArray[3 \* (positions[1] - 1) + 1];

finalBuffer[f\_count + 8] = positionArray[3 \* (positions[1] - 1) + 2];

finalBuffer[f\_count + 9] = normalArray[3 \* (normals[1] - 1)];

finalBuffer[f\_count + 10] = normalArray[3 \* (normals[1] - 1) + 1];

finalBuffer[f\_count + 11] = normalArray[3 \* (normals[1] - 1) + 2];

finalBuffer[f\_count + 12] = positionArray[3 \* (positions[2] - 1)];

finalBuffer[f\_count + 13] = positionArray[3 \* (positions[2] - 1) + 1];

finalBuffer[f\_count + 14] = positionArray[3 \* (positions[2] - 1) + 2];

finalBuffer[f\_count + 15] = normalArray[3 \* (normals[2] - 1)];

finalBuffer[f\_count + 16] = normalArray[3 \* (normals[2] - 1) + 1];

finalBuffer[f\_count + 17] = normalArray[3 \* (normals[2] - 1) + 2];

f\_count += 17;

}

}

faces = finalBuffer;

free(positionArray);

free(normalArray);

}

printf("Done Loading Object\n");

return f\_count;

}

int main()

{

float\* data = nullptr;

int faces=0;

clock\_t start = clock();

myobj\_load("stall.obj", data);

clock\_t end = clock();

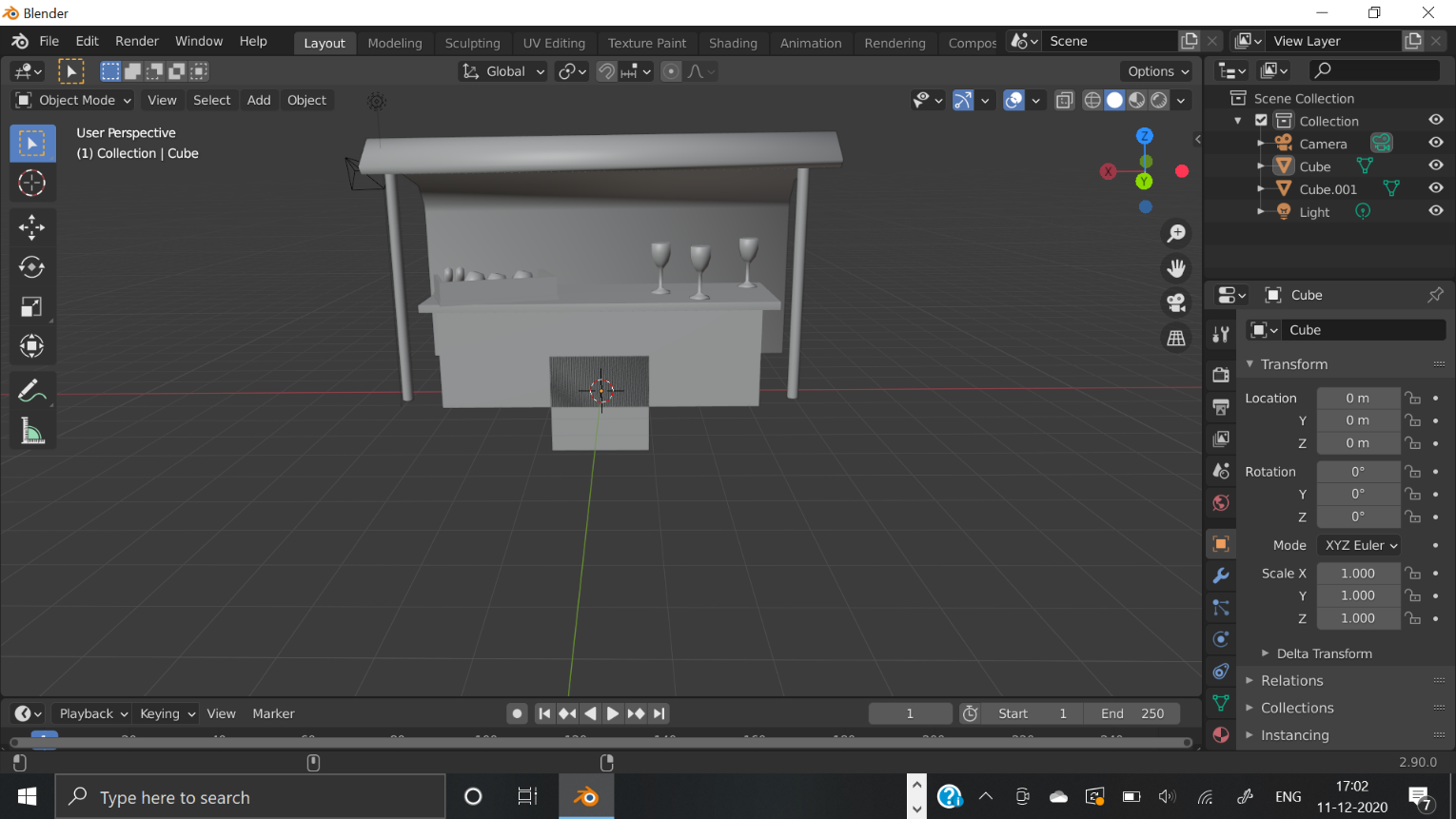
printf("Time taken: %f\n", (float)(end - start) / CLOCKS\_PER\_SEC);

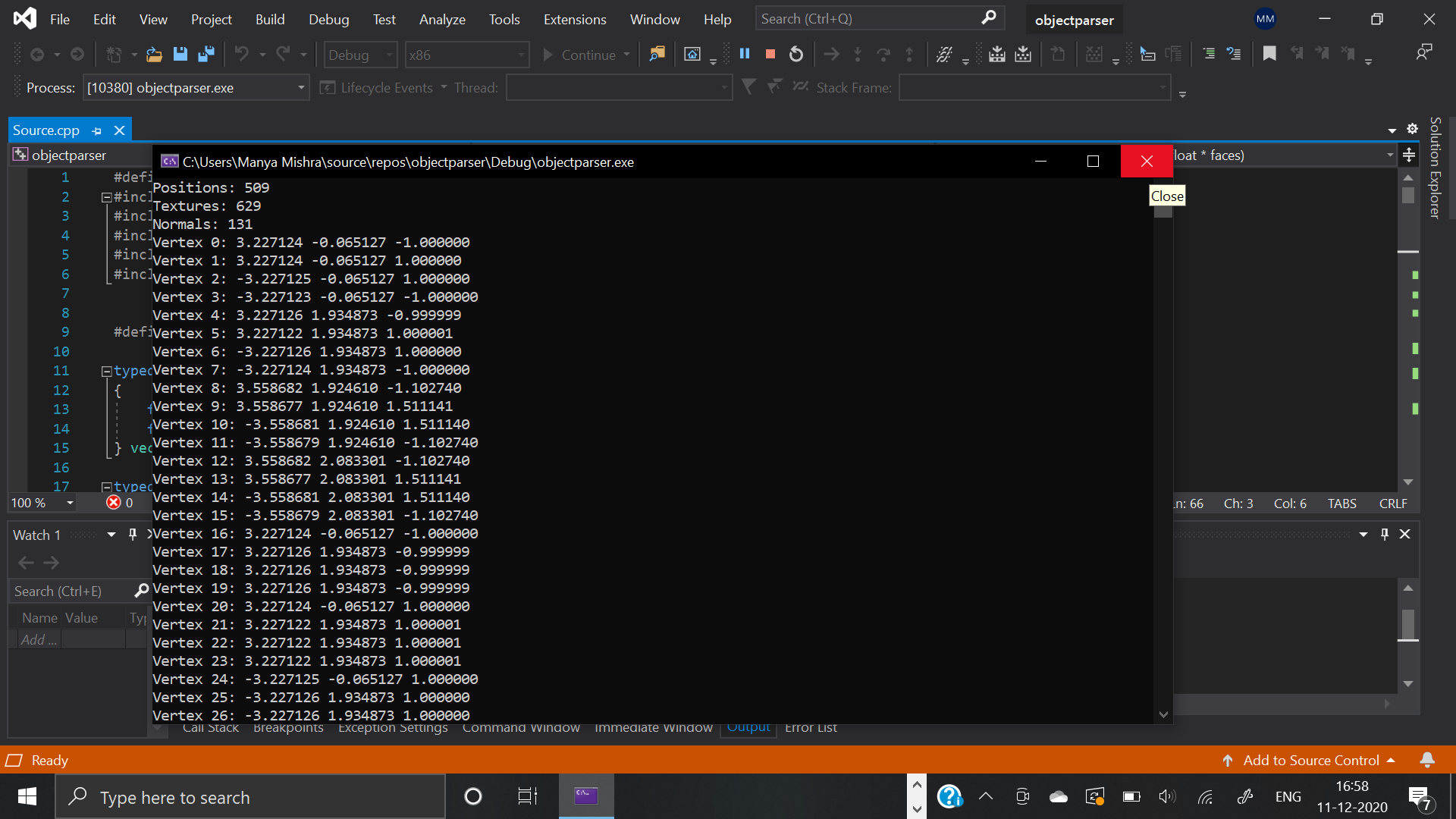
std::cin.get();

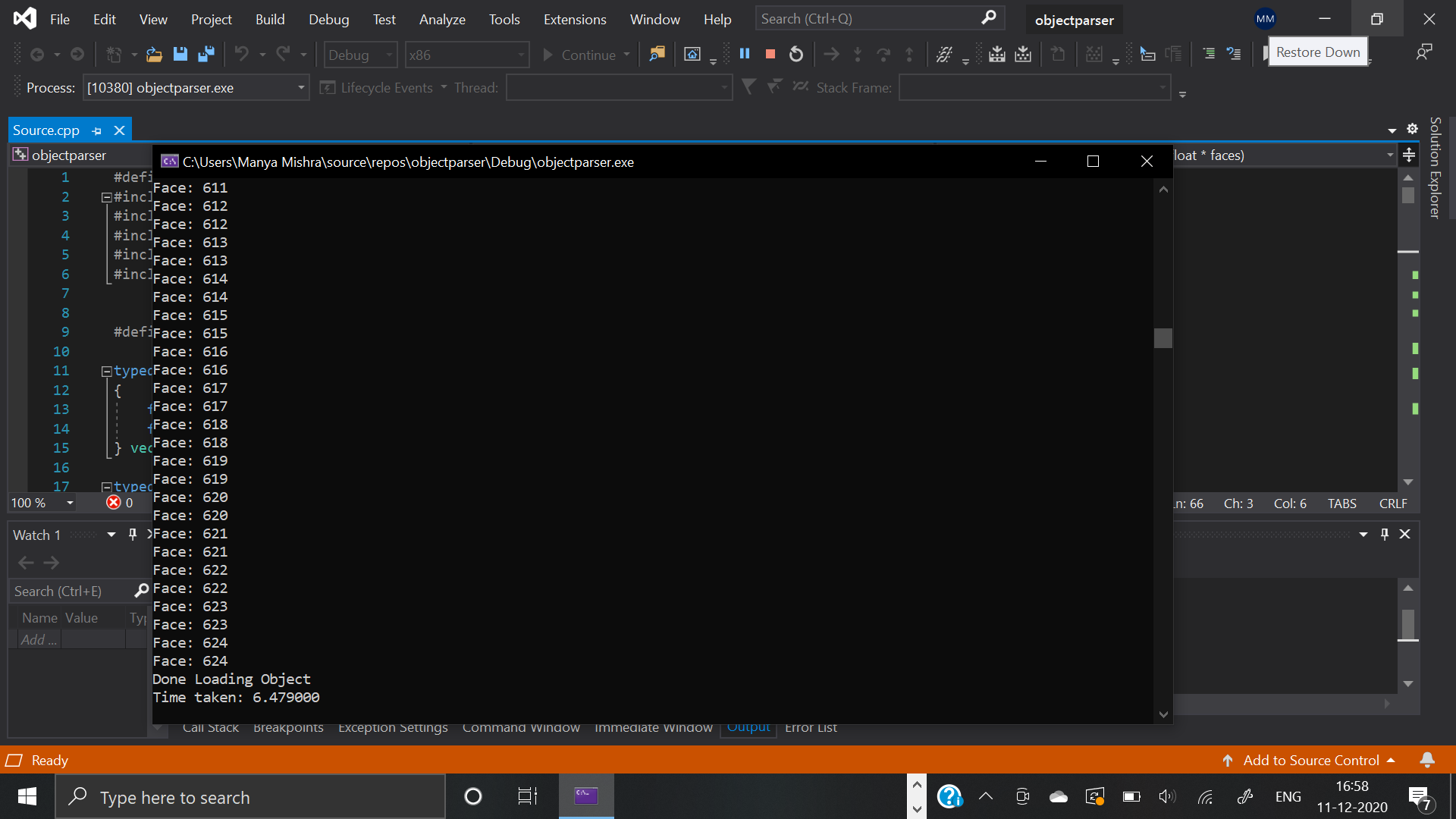
return 0;

}

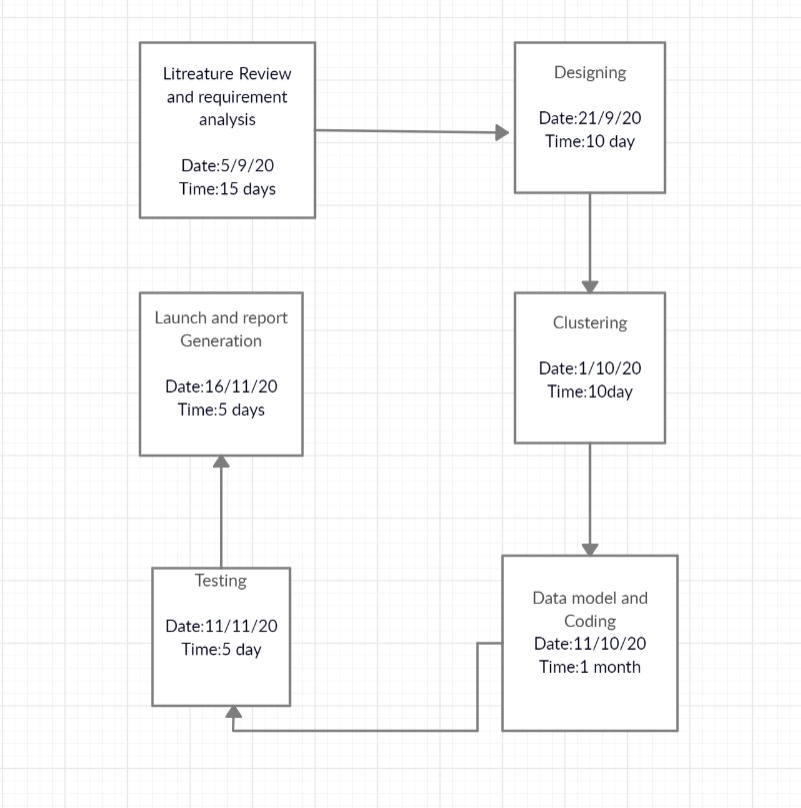
1. **Snapshot and Screenshot**







**PLAN OF WORK**

****

**Features**

* Moderate speed and moderate memory consumption
* CUBIC object (2 KB) can be loaded in 0.05 secs and a Stall object (53 KB) can be loaded in 2.3 secs.

**Current status**

* Experimental.
* Loading geometry data would be OK, more testing required for materials and shapes.

**Future Enhancements**

* **Sphere (non-standard):**  
  Spheres are defined with a position vertex and two normals: one for the up normal and one for the equator normal. The length of either normal can be chosen for the sphere radius.

v 10 10 10 vn 1 0 0 vn 1 0 0 sp 1 1 2

* **Plane (non-standard):**  
  Planes are defined with a position vertex and two normals: one for the rotation normal and one for the plane normal.

v 0 0 -10 vn 0 0 1 vn 1 0 0 pl 1 1 2

* **Light, point (non-standard):**  
  A simple point light source. Use a material definition to set the output values.

v 10 50 0 lp 1

* **Light, quad (non-standard):**  
  A 4 cornered area light.

v 0 50 0 v 10 50 0 v 10 50 10 v 0 50 10 lq 1 2 3 4

* **Light, disc (non-standard):**  
  A disc-shaped area light source. The normal specifies the direction and size of the disc.

v 0 50 0 vn 0 -1 0 ld 1 1

* **Camera (non-standard):**  
  Used to define a simple camera. The 2 vertices define the camera position and the point the camera is focusing on. The normal defines the up vector.

v 0 0 -20 v 0 5 5 vn 0 1 0 c 1 2 1

# REFERENCES

[1] <http://paulbourke.net/dataformats/obj/>

[2] <https://www.fileformat.info/format/wavefrontobj/egff.htm>

[3] <https://en.wikipedia.org/wiki/Wavefront_.obj_file>

[4] <https://threejsfundamentals.org/threejs/lessons/threejs-load-obj.html>