# Computer Science 352 Summer 2023
# Programming Assignment 8
# Due Friday, 7/28/2023 by 7pm

The 8th PA for CS 352 requires you to implement a C program for creating 3D scenes and saving them to the STL file format. The next will require you to build off of the C code you write for this one, so you should give it your best effort. The file and directory structure of the pa should be:

```
pa8
  ├── 3d.h
  ├── 3d.c
  ├── generator.c
  └── makefile
```

This homework involves writing a C library that can be used to create in-memory data structures that represent scenes in 3-dimensional space. After creating 3D scenes, the library will have the ability to save/write these scenes to a file format known as STL (Standard Triangle Language). However, the way in which you represent the 3D objects in the C library is influenced by how the STL file format works, so let's discuss that a bit before talking about how to implement the C code.

## STL Files

.stl files can be represented in binary or with regular ascii text. The text-based format uses the .stl extension, but it can be treated as regular text.

An STL text file specifies the coordinates of a sequence of 1 or more triangles in 3-D space. In 2D space, a triangle can be specified with three (x,y) coordinates, which would indicate the location of the three corners of the triangle. We refer to these corners as A, B, and C. In 3D space, a triangle is represented in basically the same way, except that we need three coordinates for each corner (x, y, z) due to the third dimension of “depth”. That is actually all that an STL file is. A specification of a bunch of 3D triangles. Here is a simple example:

```
solid scene
  facet normal 0.0 0.0 0.0
    outer loop
      vertex 0.0 0.0 50.0
      vertex 50.0 0.0 25.0
      vertex 0.0 50.0 -25
    endloop
  endfacet
endsolid scene
```

Every STL text file should begin and end with **“solid NAME”** and **“endsolid NAME”**. In-between, a STL file can have 0 or more facets, where each begins with “**facet normal 0.0 0.0 0.0**” and ends with “**endfacet**” (for now, don’t worry about the zeroes, just put them there for each facet). Within each facet you use an “**outer loop**” and “**endloop**” combination, and within specify the (x,y,z) locations of each corner using a “**vertex X Y Z**” line. You can try this out for yourself!

- Create a new PLAIN text file on your computer using vim, nano, or some other text editor
- Copy / paste the example STL into the text file
- Save it as **`sample.stl`**
- Go to https://www.viewstl.com/classic/ (or any online STL file viewer, there are others) and drag / drop the file onto the screen
- Violà! You have a REAL 3D file.

This example only had a basic triangle, but you can combine many triangles together to make more complex shapes. You're welcome to learn more about the STL file format by reading the wikipedia page, but this overview should be sufficient for you to move on to the coding part of the PA. Though not strict requirements for STL files in-general, you should use two-space indentation for this assignment and the next.

The binary format will be explained later.

## The 3D Library

The **`3d.h`** / **`3d.c`** library will provide an interface for creating a 3D scene, adding shapes to this scene (represented via a linked list (ish) data structure), and saving these scenes to **`.stl`** files so they can be viewed in a variety of other software. You will be provided with a fully-implemented header file containing the function prototypes and typedefs. Look for this on the PAs page on the course website and on D2L. You only have to implement the .c file. Also, keep in mind that you will add a few additional features to this 3d library in the next assignment. The functions you will be required to implement for this one are:

- `Scene3D* Scene3D_create();`
- `void Scene3D_destroy(Scene3D* scene);`
- `void Scene3D_write_stl_text(Scene3D* scene, char* file_name);`
- `void Scene3D_write_stl_binary(Scene3D* scene, char* file_name);`
- ```
  void Scene3D_add_pyramid(Scene3D* scene,
        Coordinate3D origin, double width, double height, char* orientation);
  ```
- ```
  void Scene3D_add_cuboid(Scene3D* scene,
        Coordinate3D origin, double width, double height, double depth);
  ```

You should also be aware that there are several custom types in use by these functions, such as **`Scene3D`**, **`Triangle3DNode`**, etc. These structs are:

- The **`Coordinate3D`** structure represents a coordinate in 3D space.

  ```
  typedef struct Coordinate3D {
    double x;
    double y;
    double z;
  } Coordinate3D;
  ```

- The **`Triangle3D`** structure represents a triangle in 3D space (three **`Coordinate3D`**s).

  ```
  typedef struct Triangle3D {
    Coordinate3D a;
    Coordinate3D b;
  ```

```
  Coordinate3D c;
} Triangle3D;
```

- The **`Triangle3DNode`** structure represents a node for a linked-list of triangles. This is needed by the program because the library will need to be able to add triangles to the 3D scene.

```
typedef struct Triangle3DNode {
  Triangle3D triangle;
  struct Triangle3DNode * next;
} Triangle3DNode;
```

- The **`Scene3D`** structure represents a full 3D scene. The **`Triangle3DNode* root`** pointer should refer to the root of a linked list representing the 3D triangles in this scene.

```
typedef struct Scene3D {
  long count;
  Triangle3DNode * root;
} Scene3D;
```

As for the function to implement:

The **`Scene3D* Scene3D_create();`** function is responsible for creating a new, empty 3D scene object. It needs to allocate heap memory for a Scene3D object and initialize the elements to 0 and NULL.

The **`void Scene3D_destroy(Scene3D* scene);`** function is responsible for freeing every chunk of allocated memory associated with this scene. It should free all memory from the **`Scene3D`** object itself, as well as the **`Triangle3DNode`** objects within the scene. It does not need to return anything.

The **`void Scene3D_write_stl_text(Scene3D* scene, char* file_name);`** function should write every object in the scene to an STL file with the name **`file_name`**. It should write the objects / triangles to the file in the order that they appear in the **`scene`** object. After you write a file, you can test if it works by opening it in an STL file viewer. Use “scene” for the name of the solid for the stl file output. The coordinates of the triangle corners in the output should be rounded to 5 decimal places.

The **`void Scene3D_write_stl_binary(Scene3D* scene, char* file_name);`** function will be responsible for writing a Scene3D to a file using the binary STL format. A binary STL file still uses the .stl extension, but writes the data to the file in a binary representation, which can result in more compact representations of triangles. This is especially important when we want to represent objects that have very high triangle counts. As a part of this PA, you’ll implement two other functions for creating shapes that can potentially require thousands of triangles to represent them. The wikipedia page for STL has a good description of how an binary STL file should be formatted, so please read this before proceeding (https://en.wikipedia.org/wiki/STL_(file_format)#Binary_STL).

The format can be summarized as:

- The first 80 bytes are considered a header. Officially, these bytes can be anything as long as it does not begin with “solid”. However, for this PA, you should set the first 80 bytes to be zeroed-out.
- The next 4 bytes should be the facet (triangle) count as an unsigned 4 byte int (**`uint32_t`**)
- After this comes the facets. Each facet should consume exactly 50 bytes.

- The first 12 bytes should be three, 4-byte little endian floats (just **`float`** on lectura) representing the normal (just 0.0 for each)
- The next 36 bytes should be the nine, 4-byte floats representing the coordinates of the corners of the triangle.
- There should be a two-byte unsigned int (**`uint16_t`**) at the end, just 0

That's basically it! After you have implemented it, test it out by writing a program to generate a binary STL, and then try opening it up in either an online STL viewer such as https://www.viewstl.com/classic/ or open with blender (https://www.blender.org/).

The **`void Scene3D_add_pyramid(Scene3D* scene, Coordinate3D origin, double width, double height, char* orientation);`** function should add a pyramid in 3D space to the passed-in scene. The parameters specify the scene to add the pyramid to, and where the pyramid should exist in 3D space. The **`origin`** represents the center of the base of the pyramid. The **`width`** represents the width / length of each side of the base of the pyramid. The **`height`** represents how tall the point of the pyramid will be (how far away the point will be from the origin). The **orientation** represents the direction that the top of the pyramid will point towards. You can expect it to be one of these strings: “forward” “backward” “up” “down” “left” “right”. Creating a pyramid will use 8 shapes in total. Four for the base, and one for each side. When we talk about these directions in 3D space, right/left refers to +/- on the X axis, forward/backward refers to +/- on the Y axis, and up/down refers to +/- on the Z axis.

The **`void Scene3D_add_cuboid(Scene3D* scene, Coordinate3D origin, double width, double height, double depth);`** function should add a cuboid in 3D space to the passed-in scene. The parameters specify the scene to add to and where the cuboid should exist in 3D space. The **`origin`** represents the center of the cuboid. The **`width`**, **`height`**, and **`depth`** represent the length on the x axis, length on the y axis, and length on the z axis, respectively. Our library (and STL files) represent shapes using only triangles, so how can one build a cuboid? Using the quadrilateral starter code, each side would have four triangles. Thus, each side of the 6 sides of the cuboid can be represented with 4 triangles, so creating a cuboid would use 24 triangles in total.

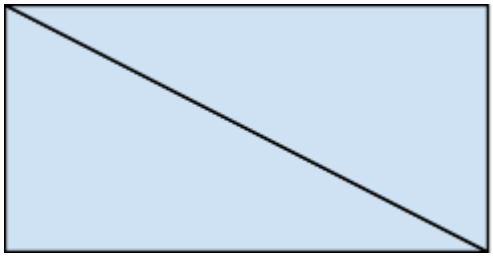
Lastly, you should implement a file named **`generator.c`**. This file should have a main function, and it may have others as well. This file should use the 3d module to create a scene with at least 10 different objects in it, and write it to an output file named **`output.stl`**. You can get creative with this - build something cool!

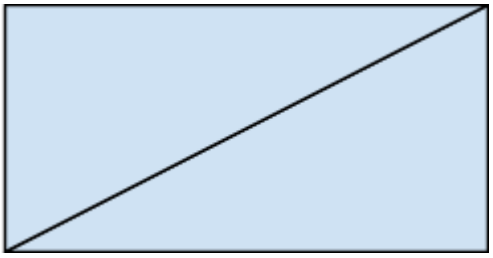
The **makefile** should produce a **`3d.o`** from **`3d.c`** object file and an executable named **`generator`** from **`generator.c`**. Running just “make” should produce **`generator`** and **`3d.o`**.

You may add additional internal helper functions in **`3d.c`**.

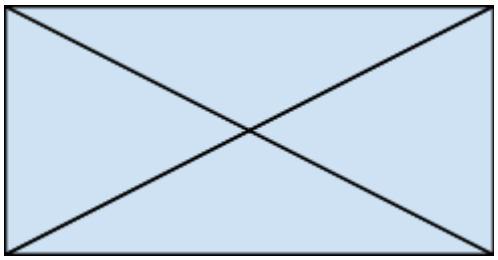
## Quadrilaterals

A quadrilateral is basically any 4-sided shape. This includes squares, rectangles, and other shapes with 4 sides. For this assignment, you will need to create 6 quadrilaterals when creating a cuboid, and one per pyramid (for the base). You will also need to create quadrilaterals for the next PA. There is a particular challenge here that I want to address. For any given quadrilateral, there are at least two ways to draw it using two triangles. For instance, below are shown two identical rectangles, but draw with the triangles at different orientations:





This inconsistency could lead to issues when it comes to testing your program. Due to this, you should draw any quadrilateral with all four triangles, rather than just two, like so:



Anytime you need to draw a quadrilateral for this PA and the next, use this function:

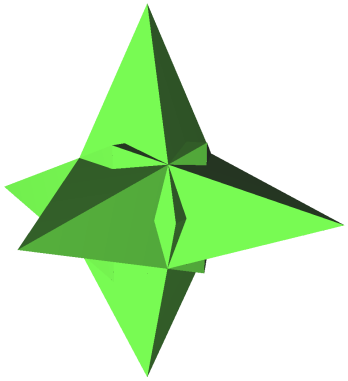
```
void Scene3D_add_quadrilateral( Scene3D* scene,
    Coordinate3D a, Coordinate3D b, Coordinate3D c, Coordinate3D d) {

  Triangle3D triangle_1 = (Triangle3D) {a, b, c};
  Triangle3D triangle_2 = (Triangle3D) {b, c, d};
  Triangle3D triangle_3 = (Triangle3D) {a, c, d};
  Triangle3D triangle_4 = (Triangle3D) {a, b, d};

  // The Scene3D_add_triangle is one you should create!
  Scene3D_add_triangle(scene, triangle_1);
  Scene3D_add_triangle(scene, triangle_2);
  Scene3D_add_triangle(scene, triangle_3);
  Scene3D_add_triangle(scene, triangle_4);
}
```

## Examples

After you have finished implementing the library, you should test to ensure that it works correctly. There are going to be a few public test-cases, which you can use to test. However, you should also try creating a few test programs of your own that build 3D scenes with multiple objects, write the STL files, and then try viewing those files to see if they are what you expect. Two examples are provided here, but you are encouraged to create your own too! The following code:

```
Scene3D* star = Scene3D_create();
char* directions[] = {"up", "down", "left", "right", "forward", "backward"};
Coordinate3D origin = (Coordinate3D){100, 100, 100};
for (int i = 0; i <= 5; i ++) {
  Scene3D_add_pyramid(star, origin, 20, 30, directions[i]);
}
Scene3D_write_stl_text(star, "star.stl");
```

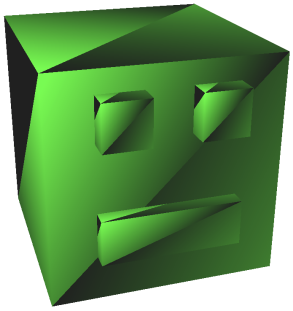
Should produce a 3D object that looks like:



And this code:

```
Scene3D* face = Scene3D_create();
Coordinate3D coordinate;
// Head
coordinate = (Coordinate3D){25, 25, 25};
Scene3D_add_cuboid(face, coordinate, 50, 50, 50);
// Eye
coordinate = (Coordinate3D){15, 40, 0};
Scene3D_add_cuboid(face, coordinate, 10, 10, 10);
// Eye
coordinate = (Coordinate3D){35, 40, 0};
Scene3D_add_cuboid(face, coordinate, 10, 10, 10);
// Mouth
coordinate = (Coordinate3D){25, 15, 0};
Scene3D_add_cuboid(face, coordinate, 30, 7, 10);
Scene3D_write_stl_text(face, "face.stl");
```

Should produce a 3D object that looks like:



Objects modeled with STL files can be printed out with a 3D printer. If you have a 3d printer, or know someone who does, you could try printing out one of the objects you create using this library!

## Submitting Your PA

Before you submit, you should ensure that:

- The file structure and file / directory names match what is required.
- The code compiles, runs, and functions correctly on Lectura.
- The code is free form memory errors and leaks.
- Remove all extra and non-required files. If you are on a mac, you should avoid zipping the file on the mac, because it might include one or more hidden / extra files. Instead, zip on lectura.
- Follow the rules from the style guide.

Once you are ready to submit, zip up the **`pa8`** directory by running:

```
$ zip -r pa8.zip ./pa8
```

Then, turn this file to the PA 8 dropbox on gradescope.

## Another example STL file with multiple triangles

```
solid scene
  facet normal 0.0 0.0 0.0
    outer loop
      vertex -25.00000 0.00000 -25.00000
      vertex 25.00000 0.00000 -25.00000
      vertex 0.00000 50.00000 0.00000
    endloop
  endfacet
  facet normal 0.0 0.0 0.0
    outer loop
      vertex -25.00000 0.00000 -25.00000
      vertex -25.00000 0.00000 25.00000
      vertex 0.00000 50.00000 0.00000
    endloop
  endfacet
  facet normal 0.0 0.0 0.0
    outer loop
      vertex 25.00000 0.00000 25.00000
      vertex -25.00000 0.00000 25.00000
      vertex 0.00000 50.00000 0.00000
    endloop
  endfacet
  facet normal 0.0 0.0 0.0
    outer loop
      vertex 25.00000 0.00000 25.00000
      vertex 25.00000 0.00000 -25.00000
      vertex 0.00000 50.00000 0.00000
    endloop
  endfacet
endsolid scene
```