M5: Capstone Final

Software Rasterizer

Stephen L. Belden

CSC1060C03 Computer Science I

August 3, 2023

Algorithm

In the interest of brevity, the written description of this program's functioning presented here is a high-level overview. The code is full of comments which provide further details.

Flow of execution begins in **main.cpp:**

> Present a menu of choices, and take action based on the user's selection.
> Load a scene and print info about the currently loaded scene.

When the render command is issued, control transfers to **scene.cpp:**

- Check that the scene loaded good data, continue if so, return to the menu if not.

- Sort triangles in the object held in the scene by depth in **sort.cpp:**
    - Calculate the depth of every triangle from the camera.
    - Insert all triangles into a list in order by depth.

- Project triangles from 3d space onto a planar 2d screen in **project.cpp:**
    - For all coordinates of all triangles, discard the axis orthogonal to the camera.

- Rasterize triangles onto a grid of pixels, and write grid to file in **rasterize.cpp:**
    - For every triangle in the list, test for intersection with every point in the grid.
        - If an intersection test returns true, color in the current pixel.
    - Print feedback to the user, as this can be a slow process.
    - After all rasterization is complete, write the result to a file:
        - Write the 54 byte BMP header.
        - Write all rows of pixels.
    - Close the file.

- Print some final statistics to the screen for the user.

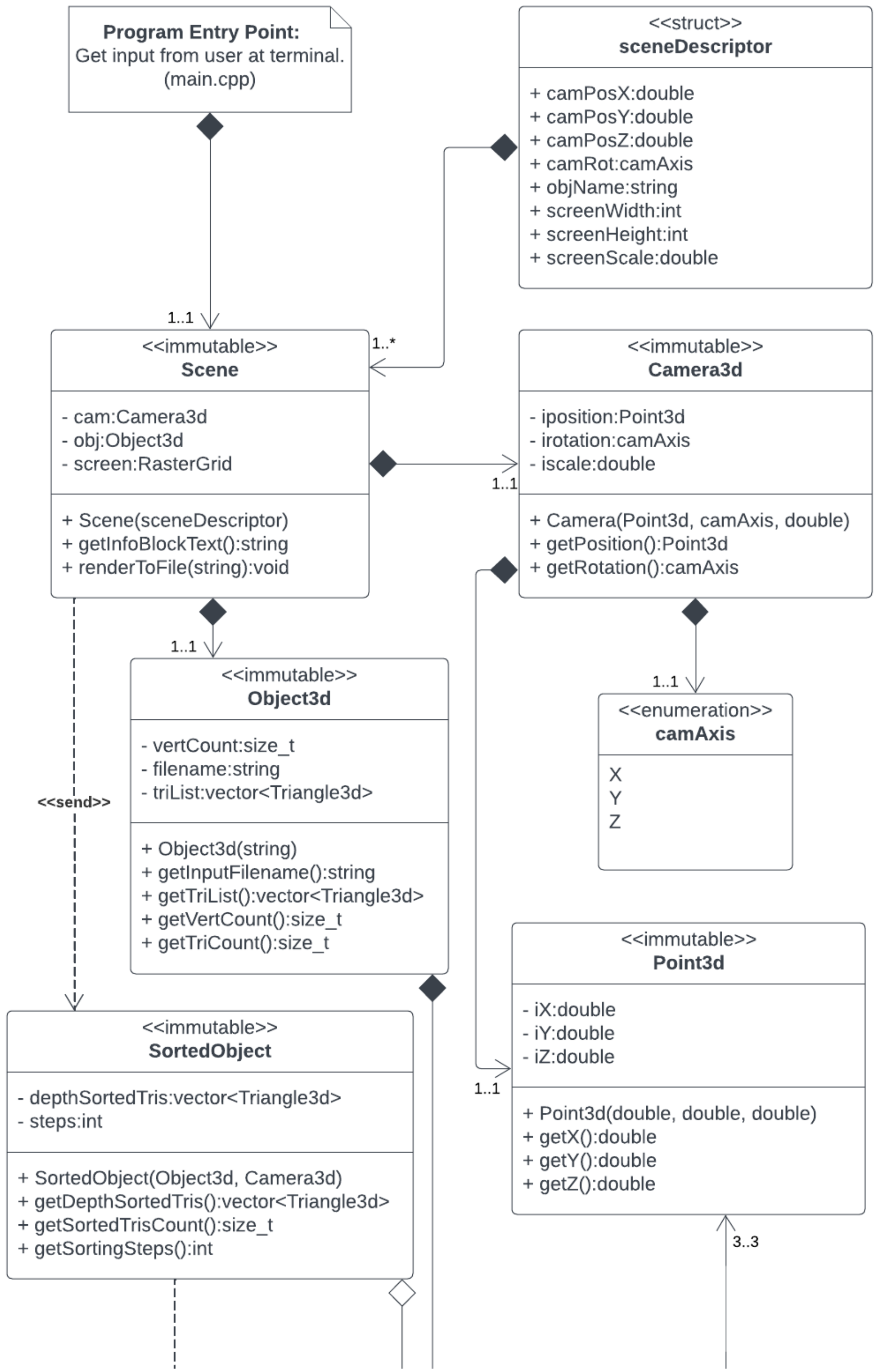
Return to the menu.

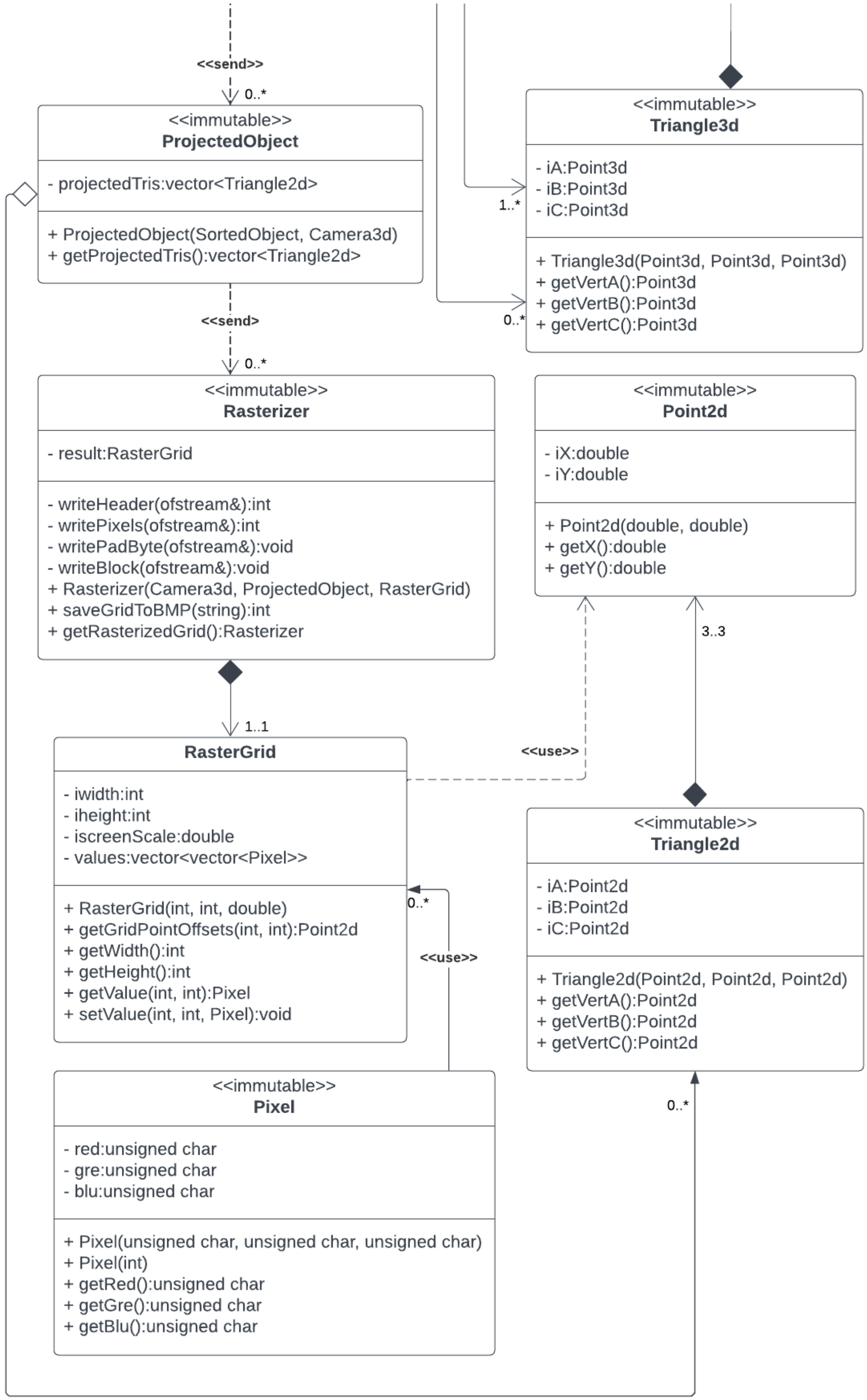
Recommendations

Finding camera settings that work well for a given .obj file requires a significant amount of trial and error. As such, I've prepared a number of preconfigured examples:

- A very simple scene of 2 triangles is loaded by default. Render it with command **R.**
    - Example output is provided in the solution directory as **tris.bmp**
- The same scene with an inverted camera can be rendered with commands **F** then **R.**
    - This demonstrates proper functioning of the depth sorting system.
    - Example output is provided in the solution directory as **tris_flip.bmp**
- Loading and rendering of larger .obj files commonly used in the graphics programming field can be done with **T** then **R** (teapot, 320x240) or **B** then **R** (bunny, 640x480).
    - This demonstrates the systems of the program at scale.
    - Examples are provided as **teapot.bmp** and **bunny.bmp** respectively.

UML


Program Entry Point:
Get input from user at terminal.
(main.cpp)
<<struct>>
sceneDescriptor
+ camPosX:double
+ camPosY:double
+ camPosZ:double
+ camRot:camAxis
+ objName:string
+ screenWidth:int
+ screenHeight:int
+ screenScale:double
1..1
1..*
<<immutable>>
Scene
- cam:Camera3d
- obj:Object3d
- screen:RasterGrid
+ Scene(sceneDescriptor)
+ getInfoBlockText():string
+ renderToFile(string):void
<<immutable>>
Camera3d
- iposition:Point3d
- irotation:camAxis
- iscale:double
1..1
+ Camera(Point3d, camAxis, double)
+ getPosition():Point3d
+ getRotation():camAxis
1..1
<<immutable>>
Object3d
- vertCount:size_t
- filename:string
- triList:vector<Triangle3d>
+ Object3d(string)
+ getInputFilename():string
+ getTriList():vector<Triangle3d>
+ getVertCount():size_t
+ getTriCount():size_t
<<send>>
1..1
<<enumeration>>
camAxis
X
Y
Z
<<immutable>>
Point3d
- iX:double
- iY:double
- iZ:double
1..1
+ Point3d(double, double, double)
+ getX():double
+ getY():double
+ getZ():double
3..3
<<immutable>>
SortedObject
- depthSortedTris:vector<Triangle3d>
- steps:int
+ SortedObject(Object3d, Camera3d)
+ getDepthSortedTris():vector<Triangle3d>
+ getSortedTrisCount():size_t
+ getSortingSteps():int



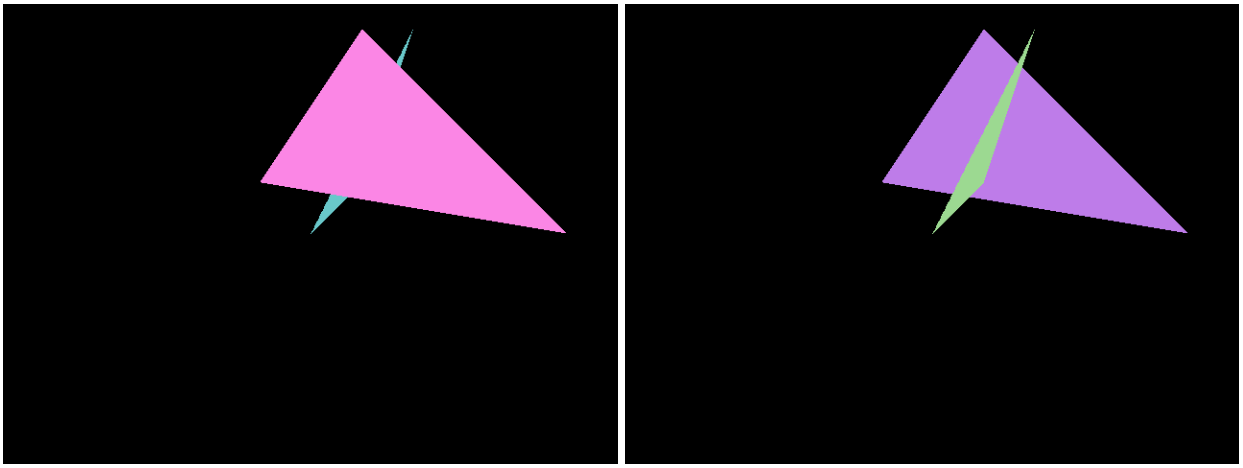
<<send>>
0..*
<<immutable>>
ProjectedObject
- projectedTris:vector<Triangle2d>
+ ProjectedObject(SortedObject, Camera3d)
+ getProjectedTris():vector<Triangle2d>
<<send>
0..*
<<immutable>>
Triangle3d
- iA:Point3d
- iB:Point3d
- iC:Point3d
1..*
+ Triangle3d(Point3d, Point3d, Point3d)
+ getVertA():Point3d
+ getVertB():Point3d
+ getVertC():Point3d
0..*
<<immutable>>
Rasterizer
- result:RasterGrid
- writeHeader(ofstream&):int
- writePixels(ofstream&):int
- writePadByte(ofstream&):void
- writeBlock(ofstream&):void
+ Rasterizer(Camera3d, ProjectedObject, RasterGrid)
+ saveGridToBMP(string):int
+ getRasterizedGrid():Rasterizer
<<immutable>>
Point2d
- iX:double
- iY:double
+ Point2d(double, double)
+ getX():double
+ getY():double
3..3
1..1
RasterGrid
<<use>>
- iwidth:int
- iheight:int
- iscreenScale:double
- values:vector<vector<Pixel>>
+ RasterGrid(int, int, double)
+ getGridPointOffsets(int, int):Point2d
+ getWidth():int
+ getHeight():int
+ getValue(int, int):Pixel
+ setValue(int, int, Pixel):void
0..*
<<use>>
<<immutable>>
Triangle2d
- iA:Point2d
- iB:Point2d
- iC:Point2d
+ Triangle2d(Point2d, Point2d, Point2d)
+ getVertA():Point2d
+ getVertB():Point2d
+ getVertC():Point2d
<<immutable>>
Pixel
- red:unsigned char
- gre:unsigned char
- blu:unsigned char
+ Pixel(unsigned char, unsigned char, unsigned char)
+ Pixel(int)
+ getRed():unsigned char
+ getGre():unsigned char
+ getBlu():unsigned char
0..*

Screenshots

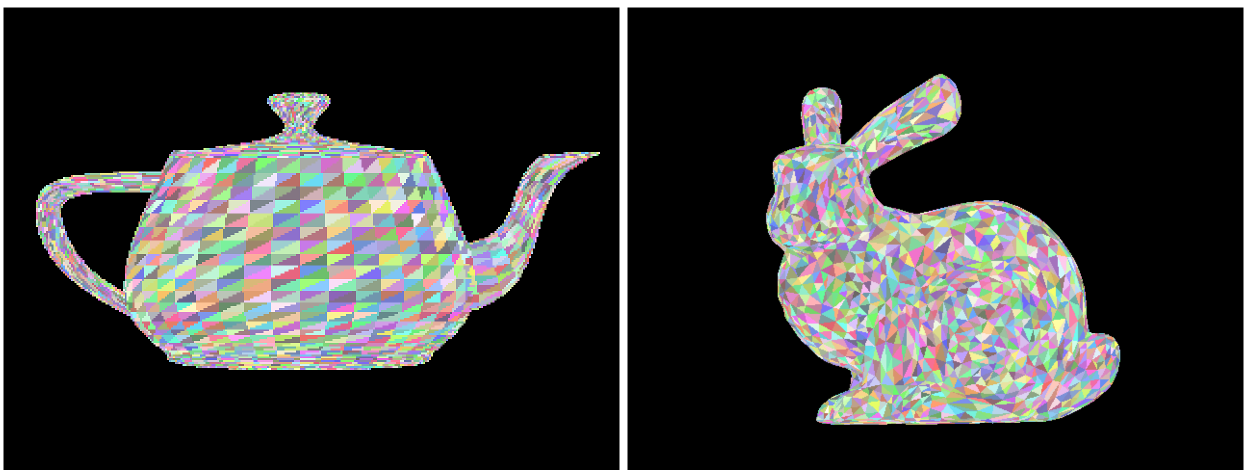
```
Enter the letter of your choice: t

######################## Scene Info ########################
Object: teapot.obj
Vertices Loaded: 3644
Triangles Loaded: 6320

Camera Position: 0.150000 1.500000 -10.000000
Camera Rotation: Z

Image Dimensions: 320 x 240
############################################################

Menu Options:
    B - load the Stanford Bunny scene (VERY slow)
    T - load the Utah Teapot scene (slow)
    S - load a Simple triangle scene (fastest)
    F - load a Flipped version of the simple scene
    C - specify a Custom scene to load
    P - Print info about the currently loaded scene
    R - Render the currently loaded scene
    E - Exit the program

Enter the letter of your choice: r

Specify output filename (without spaces): teapot.bmp

Sorted 6320 triangles by depth in 10075099 steps.
Projected 18960 3d points into 2d space orthogonal to the Z axis.
Rasterizing screen from (-3.35, -1.125) to (3.65, 4.125)
Rendered 0 out of 6320 triangles...
Rendered 250 out of 6320 triangles...
Rendered 500 out of 6320 triangles...
Rendered 750 out of 6320 triangles...
Rendered 1000 out of 6320 triangles...
Rendered 1250 out of 6320 triangles...
Rendered 1500 out of 6320 triangles...
Rendered 1750 out of 6320 triangles...
Rendered 2000 out of 6320 triangles...
Rendered 2250 out of 6320 triangles...
Rendered 2500 out of 6320 triangles...
Rendered 2750 out of 6320 triangles...
Rendered 3000 out of 6320 triangles...
Rendered 3250 out of 6320 triangles...
Rendered 3500 out of 6320 triangles...
Rendered 3750 out of 6320 triangles...
Rendered 4000 out of 6320 triangles...
Rendered 4250 out of 6320 triangles...
Rendered 4500 out of 6320 triangles...
Rendered 4750 out of 6320 triangles...
Rendered 5000 out of 6320 triangles...
Rendered 5250 out of 6320 triangles...
Rendered 5500 out of 6320 triangles...
Rendered 5750 out of 6320 triangles...
Rendered 6000 out of 6320 triangles...
Rendered 6250 out of 6320 triangles...
Rendered 6320 out of 6320 triangles.
Found 48697 pixel-triangle intersections with 485327303 misses.
Wrote 230454 bytes to file "teapot.bmp".
Files are saved in Visual Studio solution folder.
```

(Partial output from terminal interface while rendering teapot.)

Sample Output



(**tris.bmp** and **tris_flip.bmp**, showing sorting by depth reacting to camera settings.)



(**teapot.bmp** and **bunny.bmp**, showing the program's features with thousands of triangles.)

Challenges

1. Before I could rasterize and save my own renders, I needed to write a valid BMP file. This was a challenge because all data in the header must be perfectly byte-aligned. My first attempt at this produced a header with 58 bytes instead of the required 54. Careful application of a hex editor was employed to discover which variables had been written in the wrong size:

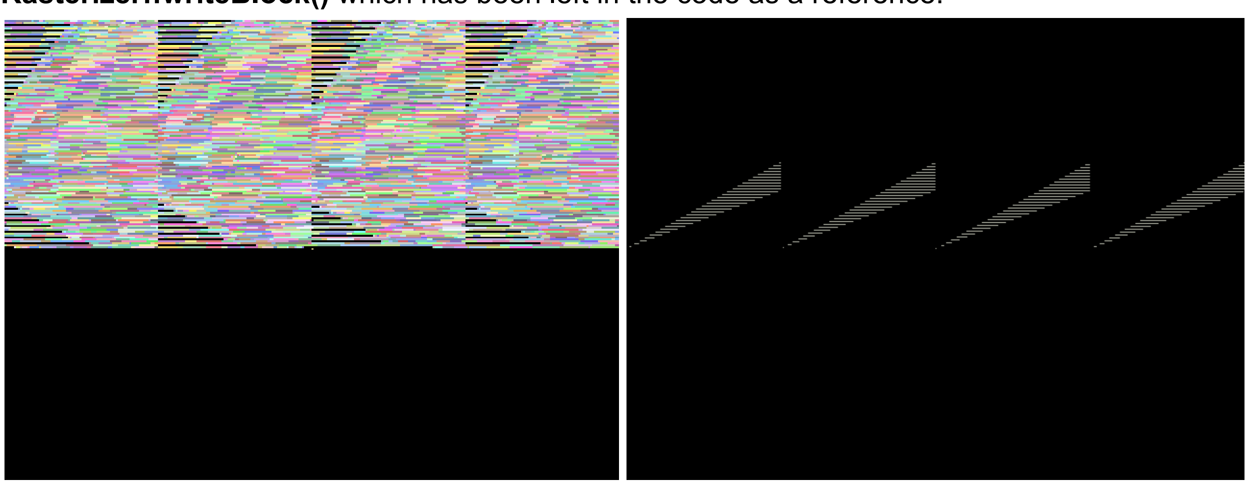
```
BROKE.bmp

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  42 4D 36 10 0E 00 00 00 00 00 36 00 00 00 28 00  BM6.......6...(.
00000010  00 00 80 02 00 00 E0 01 00 00 01 00 00 00 18 00  ..€...à.........
00000020  00 00 00 00 00 00 00 00 00 00 C4 0E 00 00 C4 0E  ..........Ä...Ä.
00000030  00 00 00 00 00 00 00 00 00 00 64 C8 FF 64 C8 FF  ..........dÈÿdÈÿ
00000040  64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64  dÈÿdÈÿdÈÿdÈÿdÈÿd
00000050  C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8  ÈÿdÈÿdÈÿdÈÿdÈÿdÈ
00000060  FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF  ÿdÈÿdÈÿdÈÿdÈÿdÈÿ
00000070  64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64  dÈÿdÈÿdÈÿdÈÿdÈÿd
00000080  C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8 FF 64 C8  ÈÿdÈÿdÈÿdÈÿdÈÿdÈ
```

(Highlighted: two 4-byte numbers which should each be 2-byte numbers.)

The code responsible for this bug wrote variables of type **uint16_t** in the size of **uint32_t**. This type mismatch could not be caught by the compiler because of the cast to **char***:

```
// Color settings
const uint16_t numberOfColorPlanes = 1; // must be 1
const uint16_t colorDepth = 24; // 3 byes per pixel
bmpFile.write((char*)&numberOfColorPlanes, sizeof(uint32_t));
bmpFile.write((char*)&colorDepth, sizeof(uint32_t));
```

2. The first fully integrated version of the project produced output that looked cool, but was not the intended result. This was caused by the accidental swapping of width and height variables in several functions. Correcting this issue necessitated the development of the testing function **Rasterizer::writeBlock()** which has been left in the code as a reference:



(My very first renders of **teapot.bmp** and a single-triangle test scene.)

References

Bourke, P. (n.d.). *Object Files (.obj)*. paulbourke.net. http://paulbourke.net/dataformats/obj/

Hassan, M. M. (2020, June 28). *C++: How to Write a Bitmap Image from Scratch.* dev.to. https://dev.to/muiz6/c-how-to-write-a-bitmap-image-from-scratch-1k6m

Newell, M. (1975). *teapot.obj*. Utah Graphics Lab. https://graphics.cs.utah.edu/courses/cs6620/fall2013/prj05/teapot.obj

Totologic: Accurate point in triangle test. (2014, January 25). Totologic. https://totologic.blogspot.com/2014/01/accurate-point-in-triangle-test.html

Turk, G., & Levoy, M. (1994, July 29). *bunny.obj*. Computer Graphics at Stanford University. https://graphics.stanford.edu/~mdfisher/Data/Meshes/bunny.obj