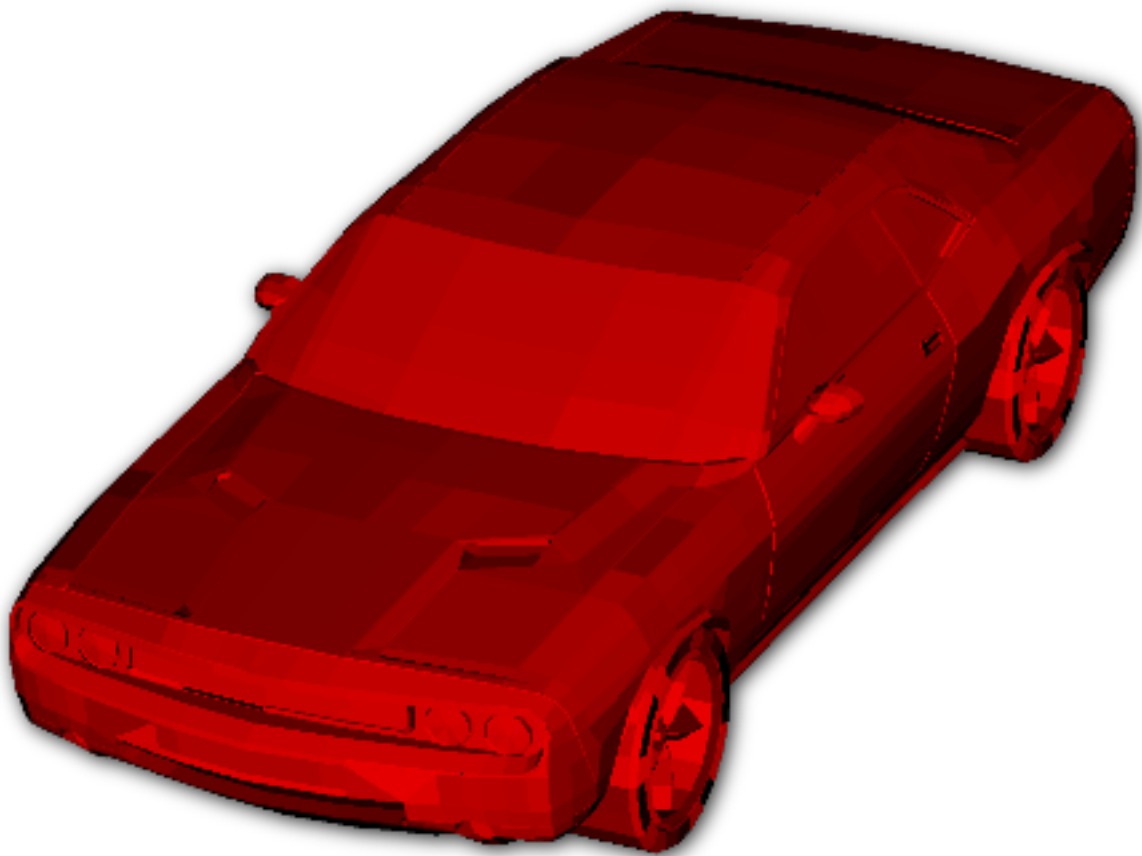


Project LUX



Analysis	5
Introduction	5
Research	6
Identification of end users	6
Requirements	6
Acceptable limitations	8
Possible data structures for space partitioning	8
Octree	8
Binary Space Partition (BSP) tree	9
Bounding Volume Hierarchy (BVH)	10
My choice	11
Possible programming languages and graphics libraries	12
C++ with SDL2	12
Python 3 with Pygame	12
My choice	12
Possible file formats for 3D model storage and loading	13
STL (STereoLithography) format	13
Wavefront OBJ format	13
My choice	14
Possible linear algebra libraries	14
BLAS	14
LAPACK	15
Implement it myself	15
My choice	15
Possible GUI frameworks	16
Qt in C++	16
Tkinter in Python 3	16

My choice	16
Design	17
High-level overview	17
Class Relation Diagram	18
User Interface (HCI)	19
Hardware	21
Data Structures	21
OBJ files	21
Bounding volume hierarchy (BVH)	22
Indexed vertices	23
Vectors and matrices	24
Rays	25
Models	25
Algorithms	26
Ray-triangle intersection	26
Ray-sphere intersection	29
Partition triangles	31
OBJ file parser	33
Reinhard tone-mapping operator	34
Sending data from Python to C++	36
Python	36
C++	36
Classes	37
Camera	37
BVHNode	39
Model	41
Triangle	42

Ray	44
Vec3	45
Mat3	46
Non-class functionality	47
ModelLoader	47
GUI classes	48
MainWindow	48
ModelWindow	51
ModelData	53
Data dictionaries	54
Python to C++ communication	54
Technical solution	55
C++	55
module.cpp	55
camera.h	61
camera.cpp	62
model.h	66
model.cpp	68
modelloader.h	72
modelloader.cpp	73
BVH.h	76
BVH.cpp	78
transform.h	83
transform.cpp	84
geometry.h	86
geometry.cpp	88
Python	92

setup.py	92
main.py	93
mainwindow.py	93
modelwindow.py	100
modeldata.py	105
Testing	107
Renderer testing	107
GUI testing	112
Main	112
MainWindow	112
ModelWindow	113
ModelData	114
Testing images	115
Renderer	115
GUI	140
Evaluation	150
Fulfillment of requirements	150
Overall conclusion	151
Examples of final renders	152

Analysis

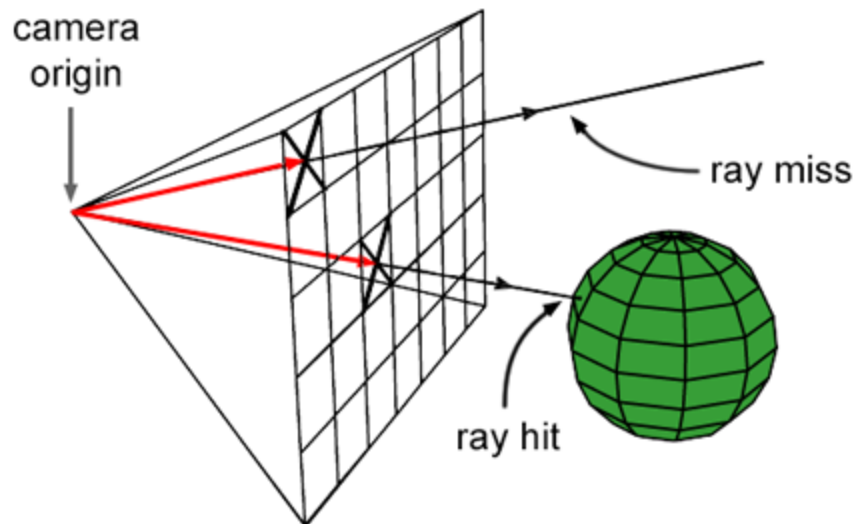
Introduction

For my project I will be writing a triangle-based 3D renderer.

A 3D renderer takes 3D models in a scene and draws them onto the 2D surface of the screen from a perspective viewpoint.

One of the main techniques for rendering is called ray-tracing, where 'rays' are emitted into the scene per pixel and their collisions with geometry are calculated, thus giving information about the colour of that pixel.

You could imagine that an image of the 3D scene is being projected onto the 2D plane (i.e. the screen).



I chose this technique over other alternatives (such as the more popular scanline triangle rasterisation) because of how accurately it models the real world, representing actual rays of light. The only conceptual difficulty is the fact that rays are traced in the opposite direction they would travel in real life, being traced from the eye outwards to the light source.

Triangle meshes are used to represent models, as while a ray-tracer can render many other shape primitives, I again wanted to streamline the program. As well as this, with a high enough density triangle meshes can approximate any shape.

Typically each emitted ray must test for collision with every triangle in the scene, which can be incredibly time consuming and wasteful. To alleviate this I will store the triangle meshes in an

acceleration structure. These spatially divide the set of triangles into subsets that can be recursively tested for, allowing pruning to minimise unnecessary calculations.

Research

I have researched the problem and possible solutions mostly from online articles and research papers. My knowledge of the subject area has been steadily accrued over a number of years through many videos, articles, and projects; so my list of sources is fragmented at best.

Real-Time Collision Detection for Dynamic Virtual Environments

http://www-ljk.imag.fr/Publications/Basilic/com.lmc.publi.PUBLI_Inproceedings@117681e94b6_1860ffd/bounding_volume_hierarchies.pdf

My main source for how bounding volume hierarchies work. Describes many different bounding volumes with their pros and cons, as well as implementation details. Also includes details on how a BVH could be adapted for representing a scene of moving objects.

Ray-Triangle Intersection Algorithm

<https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection>

An explanation of the Möller-Trumbore algorithm, for computing the intersection point of a ray and a triangle. The derivations of the various formulae required are clear, as well as a concise description of the barycentric coordinate system. The source is also included, written in C++.

Ray-Sphere Intersection Algorithm

<https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection>

A description of two algorithms for computing the intersection points between a ray and a sphere. One is a geometric solution, the other an analytic solution. Both are comprehensively explained, and as above, the source code is included, written in C++.

Identification of end users

This is a research project, and so there is no end user, the purpose is to explore how ray-tracing can be implemented in software.

However while there is no defined end user, others may want to use my project as a starting point to extend its functionality into new areas, so clarity of code and ideas are important.

Requirements

The program must be able to:

- Provide a graphical user interface.
 - Allow customisation of variables such as dimensions of the window, and camera attributes such as position and field of view.
 - Allow a variable number of models to be added, and their positions and orientations in world space to be specified.
 - Allow the renderer to be started from the GUI, and the user-defined variables must be sent to the program to be used during execution.
 - Check the validity of inputted data, and notify the user if invalid, also preventing the renderer from starting.
- Parse triangle meshes from externally generated 'OBJ' files.
 - Read files line by line.
 - Parse vertex and normal data into contiguous lists.
 - Parse indexed face data into separated 'Triangle' objects.
- Store the triangle meshes in BVHs, one per model.
 - Vertices and normals are stored by the model itself
 - Triangle objects should be stored in a contiguous array in the BVH
- BVHs should recursively spatially partition the triangles.
 - Triangles should be split by axis-aligned planes to create evenly sized, spatially distinct subsets.
 - Subsets must be evenly sized to create a balanced tree to ensure a low worst case search time.
 - Subsets must be spatially distinct to minimise overlap between bounding volumes, to improve the overall efficiency of ray intersection calculations.

- Render models to the screen by emitting rays, one per pixel, and testing for collisions with geometry using the BVHs for efficiency.
 - Rays are emitted from a camera origin, through a projection plane representing the screen in world space.
 - Rays test for collisions with bounding volumes recursively.
 - If a bounding volume is successfully intersected with, test for intersections with its children.
 - If leaf node bounding volumes are intersected, intersection with triangle primitives is calculated.
 - Iterate through the list of triangles in the leaf node.
 - If an intersection is found, return the distance to the intersection point.
 - When multiple triangles overlap the same pixel, use the triangle closest to the camera.
 - Do this by comparing 't' values, the distances to the intersection points. The lesser the distance, the closer to the camera it is. The triangle with the least distance gets drawn to that pixel.
 - Triangle back faces should be ignored.
 - Compute dot product of face normal and ray direction, negative values means the front face is visible, positive values mean the back face is visible.
 - If no triangles are intersected with, set the pixel to some defined 'background' colour.
- Triangles must be rendered with directional shading to identify things like curved surfaces, otherwise they'd be shown as a silhouette of uniform colour.
 - Triangles facing the light source should be brighter than those facing away.
 - There should be a smooth transition from light to dark (on a per triangle basis)
- Multiple models must be able to be rendered in the same scene, with correct ordering so that models are not rendered over other models in front of them.

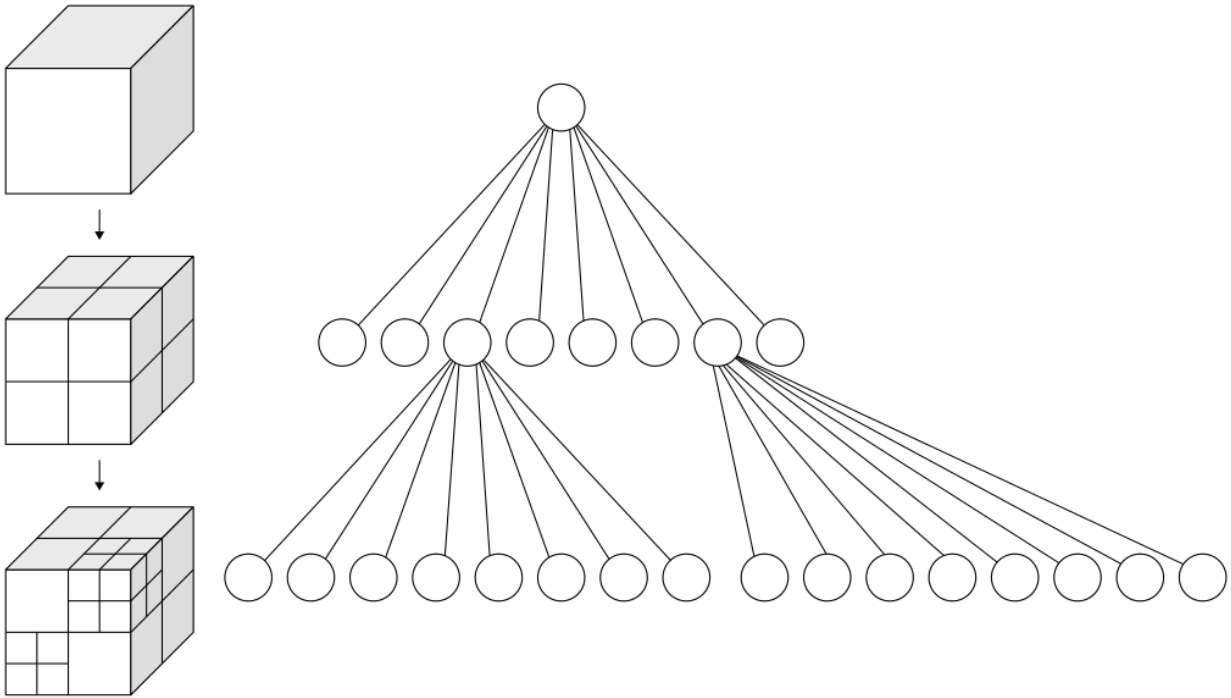
Acceptable limitations

- The shading will be done per face without interpolated normals, and there will be no advanced effects or post-processing such as shadows, smooth shading or ambient occlusion.
- The program will be run on the CPU. I will not use explicit multithreading or the GPU to accelerate processing.
- There will be no texture mapping, so texture UV coordinates do not need to be included in the OBJ files.
- Models in the scene will be stored in a flat list rather than in a scene-level bounding volume hierarchy. I will not be rendering scenes with especially large numbers of models, but if others wish to, the model-level BVH can be easily retrofitted for the scene-level to contain models instead of triangles.

Possible data structures for space partitioning

Octree

An octree partitions 3D space by recursively dividing it into 8 octants. The 3D space is represented by cuboidal bounds. These bounds are aligned to the x, y, z axes.



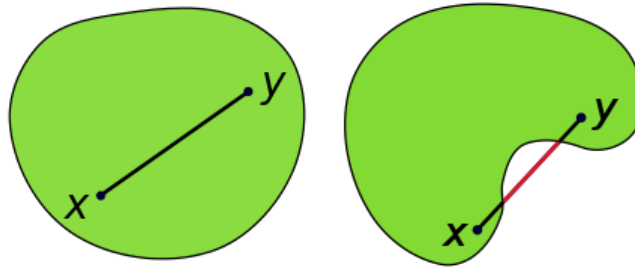
The space is subdivided based on the number of objects within it. If it's over some maximum then the space is subdivided. As each node can produce 8 additional nodes, the size of the tree can grow very large.

If objects are clustered very close together, then the region will recursively subdivide many times until it is small enough to split the set. This may use lots of unnecessary data, as each internal node has 8 children, and adversely affect performance as when calculating intersections, the ray must traverse all the way down. The tree will be implemented using pointers, so this could affect performance by indirection.

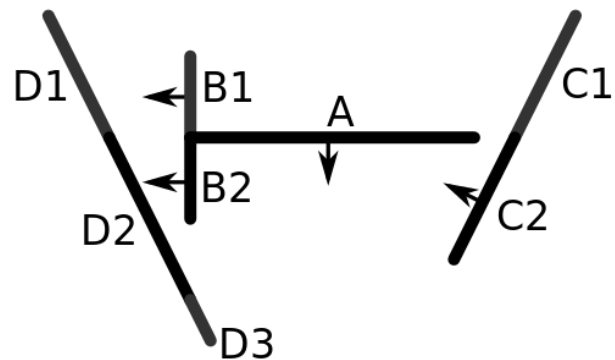
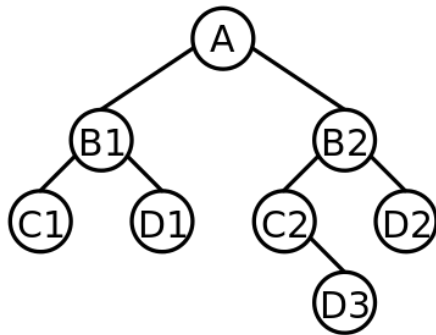
Octrees are most useful for efficiently determining collisions between dynamic objects within the octree, for example by nearest neighbour, and an octree is quick to alter as objects move. However my project will use rays that stretch infinitely, so keeping them within the octree is impossible, and so the nearest neighbour technique that is usually so effective, is not possible for me to use.

Binary Space Partition (BSP) tree

A BSP tree recursively subdivides the space into convex regions by hyperplanes (2D planes in my 3D case). These hyperplanes can have arbitrary orientation, contrasting with octrees. Convex regions are regions where for every combination of two points, the line between the points never passes outside the region.



BSP trees are advantageous because of their correctness, as no objects cross split planes. This assuredness comes at the cost of having to find the optimal splitting planes with an arbitrary orientation, which is complicated to do, and takes much longer to do than if non-optimal splitting planes were used instead.

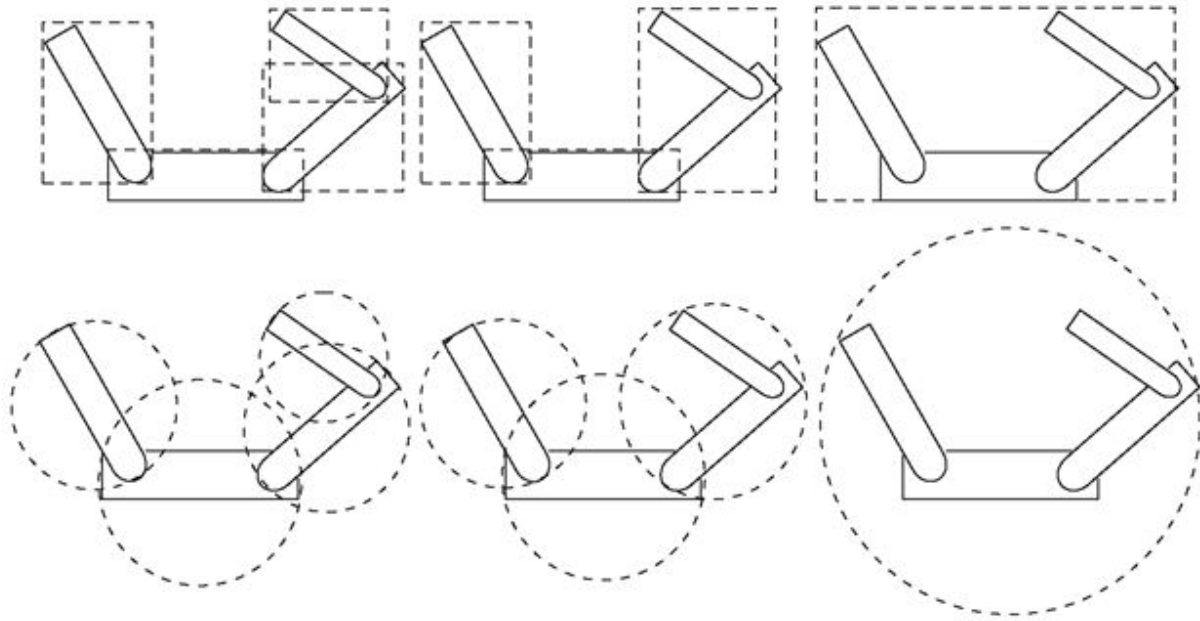


This requirement of correctness means that geometry may have to be split into smaller pieces than it already exists, increasing the polygon count and thus the execution time.

Another advantage is that a BSP tree allows a set of unordered polygons to be perfectly rendered from back to front from any camera position without having to do any actual sorting. However this only works when rasterising triangles directly, so since I am rendering per pixel instead of per polygon, determining overlap with a 'distance to intersection' value, this won't affect my project.

Bounding Volume Hierarchy (BVH)

A BVH recursively subdivides a set of objects into spatially local groups, and places a bounding volume around the extents of the set. This is done repeatedly until the subsets are small enough.



Bounding volumes are recursively tested against for collisions, instead of each element of the subset individually, increasing speed.

BVHs are designed to calculate collisions with objects external to the BVH itself, and are fast at doing so. This is very beneficial for me, as rays can be considered external objects.

Bounding volumes can also be any arbitrary shape, all the way from spheres to discrete oriented polytopes. The more complicated a bounding volume is, the more expensive it is to check for collisions, but the tighter it fits to it's set of triangles.

My choice

I will use BVHs to store model data, as they are particularly effective for ray-tracing collision detection, being designed partly for the task.

Using octrees would take more memory, and is far better suited to detecting intersections between objects within the octree itself, rather than external objects.

Using BSPs would take a long time to build, due to the required correctness and potential polygon tessellations. Their main advantage is being able to render back to front without sorting, but I will already have a solution for that.

Possible programming languages and graphics libraries

C++ with SDL2

I have reasonable experience with C++, but none with SDL2. C++ is difficult to use well, due to it being a relatively low level language, and gives some degree of control over things like memory.

However these drawbacks are countered by the speed of C++ code, as it compiles to a binary executable rather than bytecode interpreted by a virtual machine. The control over memory would allow me to know the exact size of objects, control pointer and reference passing, as well as allocate large contiguous blocks of memory for fast traversal.

SDL2 is a lightweight library that can be customised to include only the features necessary, which would be good for my project as I will only be using a graphics library to put individual pixels to the screen.

Python 3 with Pygame

I have a lot of experience with both Python and Pygame, as it's very easy to get a new project up and running, and to quickly experiment with new ideas. However in my testing of Pygame I have found large performance problems when the number of elements being drawn gets too large, especially when it must be done per frame.

This is also a drawback of Python, as it runs using an interpreter rather than compiling to a binary executable, and is a very high level language. While this gives a lot of flexibility for mistakes in terms of type checking and other things, the level of book-keeping drastically affects performance.

While my project is not tightly bound by performance, only drawing a single image to the screen a pixel at a time, having some measure of speed will make the iterative testing process much less time consuming, as I can spend more time fixing bugs than watching the image slowly render.

My choice

I will choose C++ with SDL2, as while I have more experience with Pygame than SDL2, the language chosen is far more important, as the graphics library will only be used for putting individual pixels to the screen. While Python is easier to use than C++, C++'s extra speed will make this tradeoff worth it.

While speed is not a critical consideration, it will be very nice to have.

Possible file formats for 3D model storage and loading

STL (STereoLithography) format

STL is a very simple, lightweight format that was designed for use in 3D printers (hence the name). It contains only vertex and normal data, without any representation of colour or texture coordinates. This simplicity is a benefit, however it stores every triangle as an independent sequence of 3 vertices, rather than allowing for indexed vertices, which can save considerable space.

STL can be either in a human-readable ASCII text format or a binary format. The ASCII format is rather verbose, for example each individual triangle is represented by:

```
facet normal  $n_i$   $n_j$   $n_k$ 
  outer loop
    vertex  $v1_x$   $v1_y$   $v1_z$ 
    vertex  $v2_x$   $v2_y$   $v2_z$ 
    vertex  $v3_x$   $v3_y$   $v3_z$ 
  endloop
endfacet
```

For a model of hundreds or thousands of triangles, this leads to files that are impractical to read for a human, and unnecessarily large.

The binary format is much more compact, but is obviously not human-readable, which is a requirement for me for debugging purposes.

Wavefront OBJ format

OBJ is a more powerful format than STL, allowing for storage of colour and texture coordinate information. While I won't be using these for my project, others who want to improve on my work may find it useful to have that flexibility already in place, rather than having to convert to an entirely new format.

OBJ stores a sequence of unique vertices and normals. Defined triangles index these vertices and normals using stored indices, and the same vertex can be indexed multiple times. This can potentially use a fraction of the storage space than if each triangle stored its vertices by value, such as in the STL format.

The OBJ format is compact while still being human-readable, separated logically into vertices, normals, and faces. For example this is an OBJ file representing a cube:

```
# Blender v2.79 (sub 0) OBJ File: ''
# www.blender.org
o Cube
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
v 1.000000 1.000000 -0.999999
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vn 0.0000 -1.0000 0.0000
vn 0.0000 1.0000 0.0000
vn 1.0000 -0.0000 0.0000
vn 0.0000 -0.0000 1.0000
vn -1.0000 -0.0000 -0.0000
vn 0.0000 0.0000 -1.0000
s off
f 2//1 4//1 1//1
f 8//2 6//2 5//2
f 5//3 2//3 1//3
f 6//4 3//4 2//4
f 3//5 8//5 4//5
f 1//6 8//6 5//6
f 2//1 3//1 4//1
f 8//2 7//2 6//2
f 5//3 6//3 2//3
f 6//4 7//4 3//4
f 3//5 7//5 8//5
f 1//6 4//6 8//6
```

My choice

I will use the Wavefront OBJ format, as it both compact and human-readable, taking the best of both worlds from the STL ASCII and binary formats.

Possible linear algebra libraries

BLAS

BLAS (**B**asic **L**inear **A**lgebra **S**ubroutines) provides routines for highly optimised vector and matrix operations. Level 1 implements vector-vector operations, level 2 adds vector-matrix operations, and level 3 adds matrix-matrix operations. As I will be using matrix-matrix operations, I will need to use BLAS level 3.

BLAS is very lightweight and portable, only including necessary functionality, so the size of the executable would be kept small.

User testing¹ shows up to 10 times performance gains, as BLAS utilises the SSE (streaming SIMD extensions) instruction set to parallelise operations on vectors, and optimises cache utilisation.

LAPACK

LAPACK (**L**inear **A**lgebra **PACK**age) is a wrapper for BLAS that composites the low level operations to create more general linear algebra routines, such as solving systems of linear equations. As this comes up in my program (specifically in the ray-triangle intersection), LAPACK would be a good choice.

However it would also implement a lot of functionality that I don't need, which would increase the size of the binary executable unnecessarily.

Implement it myself

Implementing linear algebra operations myself would give me a very good understanding of how they work. Even though they will be an order of magnitude slower than a highly optimised library, my project is not about rendering fast, it's about understanding how the rendering process works.

My choice

As this is a research project, I will implement the linear algebra routines myself, to learn how they work more effectively than if I used a library that does it for me. Using a library would make these operations a black box that I wouldn't understand, only use. It will also give me a good idea of how they should be implemented and optimised.

¹ <https://stackoverflow.com/questions/1303182/how-does-blas-get-such-extreme-performance>

Possible GUI frameworks

Qt in C++

Qt is an extensive and well tested framework that comes with the 'Qt Creator' visual development environment, including a visual debugger. This makes development very quick and easy once Qt is installed.

However I am using Visual Studio for developing the project, and Qt is quite difficult to integrate into a VS project, especially as it uses its own specialised IDE.

Tkinter in Python 3

Tkinter is a barebones framework that includes only the essentials, which is all I need.

The main problem with using Tkinter is that I would have to use Python for the GUI, while my main application will be written in C++ with SDL2, so I will have to be able to pass information from a Python application to a C++ one.

This is possible by exporting the C++ project as a dynamic library that is imported by the Python application.

My choice

I will use Tkinter in Python 3, as it is much easier to install and get running, and developing with a Python based framework will be easier than in a C++ based one.

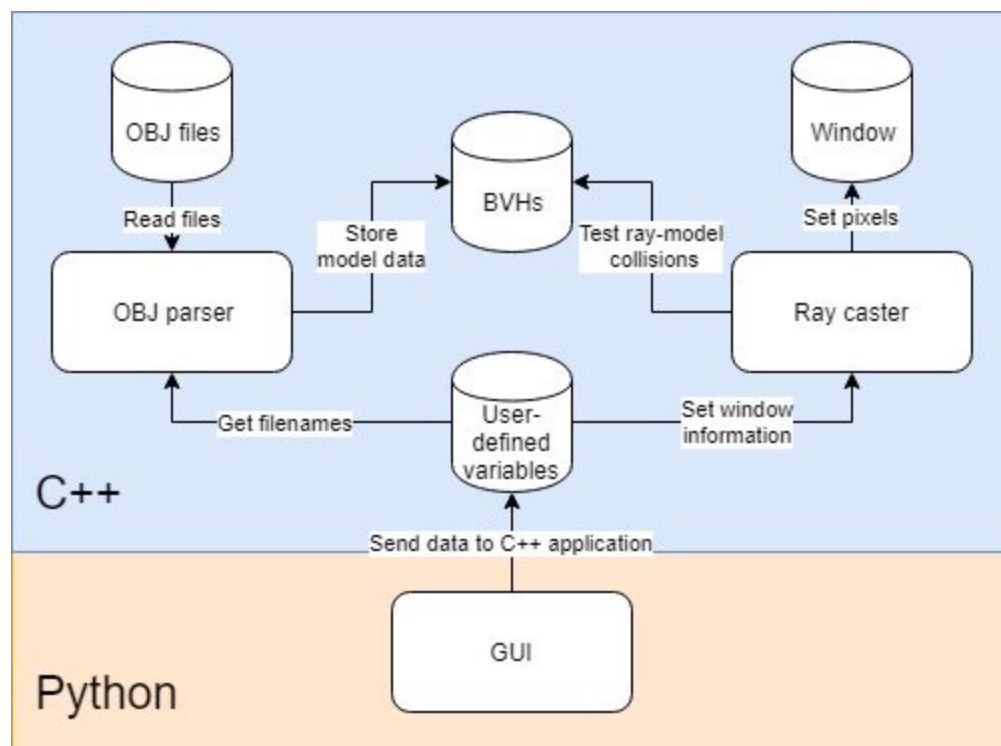
Design

High-level overview

Within the C++ section the SDL2 graphics library will be used to open and manage a window, and to set the colours of individual pixels on the screen. Everything else will be implemented by myself.

In the Python section Tkinter will be used to provide a GUI, and allow the user to customise variables.

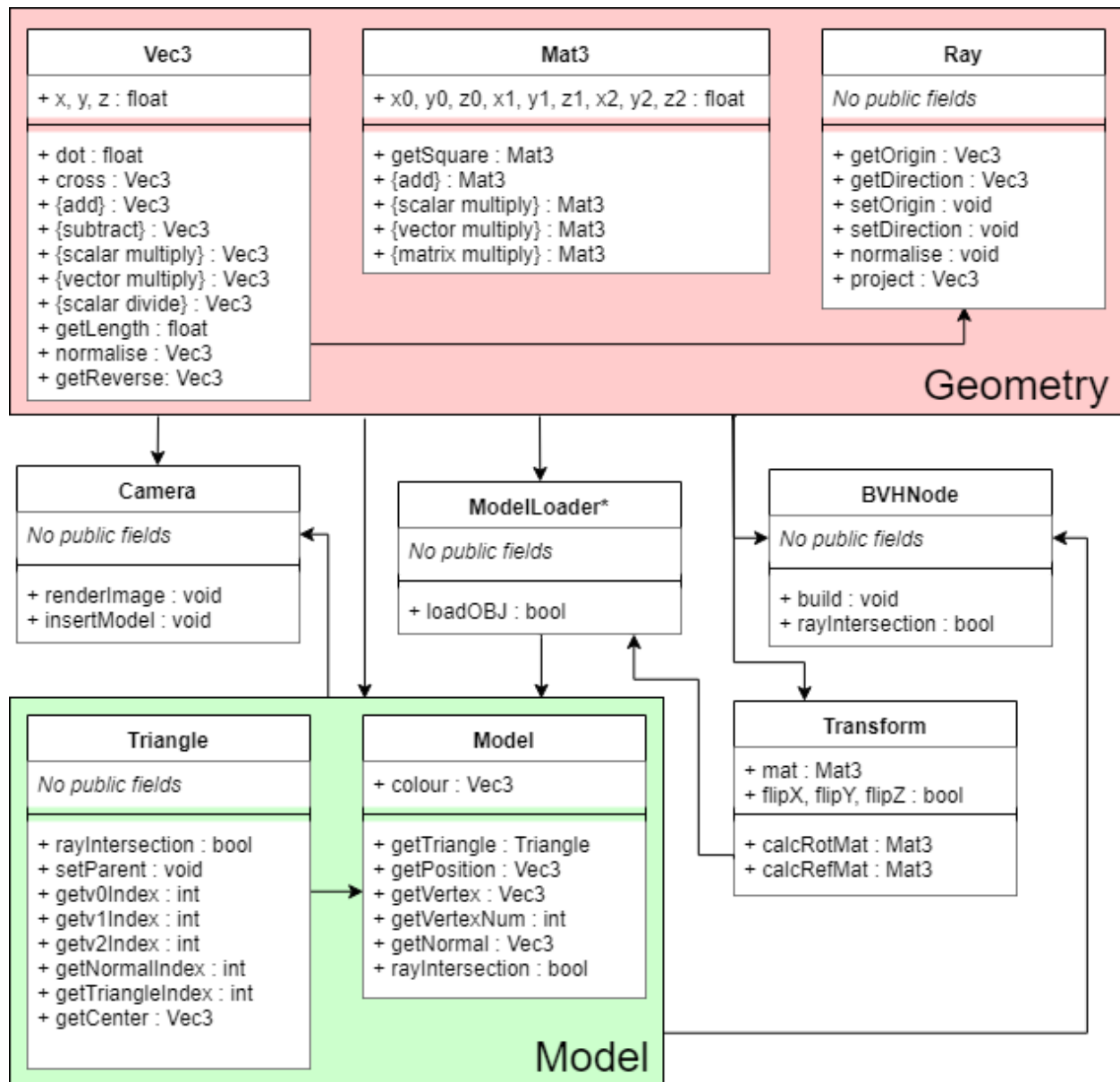
It will be written with a thin object-oriented wrapper over the event-driven Tkinter interface. The object-oriented part serves only to encapsulate distinct behaviour and data.



The C++ section is separated into two parts, the construction of the BVHs from 'OBJ' file data, and the rendering to the screen using the BVHs. The first part produces the BVHs, the second part uses them.

It will be written almost entirely object-oriented.

Class Relation Diagram



Only public fields and methods are shown, as private members do not matter for relationships between classes. The coloured boxes group together classes that are tightly coupled. As these classes will be commonly used together, it's more useful to think of them as a single entity. Member functions in curly braces are overloaded operators.

(*): ModelLoader is not a class, but for the purposes of the class relation diagram it is useful to imagine it as if it were.

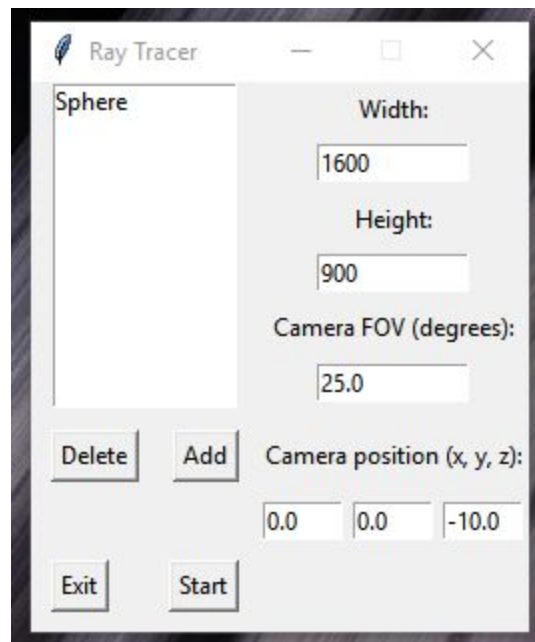
User Interface (HCI)

The user will interact with the program using a Tkinter GUI. The variables to be customised are:

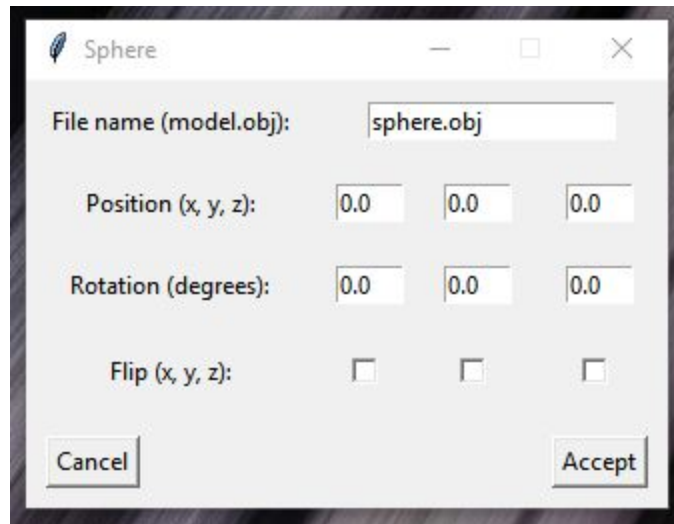
- The width and height of the window in pixels
- The position of the camera in the world, in x, y, and z coordinates, and the camera's field of view in degrees
- The models that will be rendered.
 - Their filename
 - Their position in x, y, and z coordinates
 - Their rotation in the x, y, and z axes in degrees
 - Whether they will be mirrored in the x, y, and z axes

The user can also quit the GUI, or start the renderer using the variables they have defined.

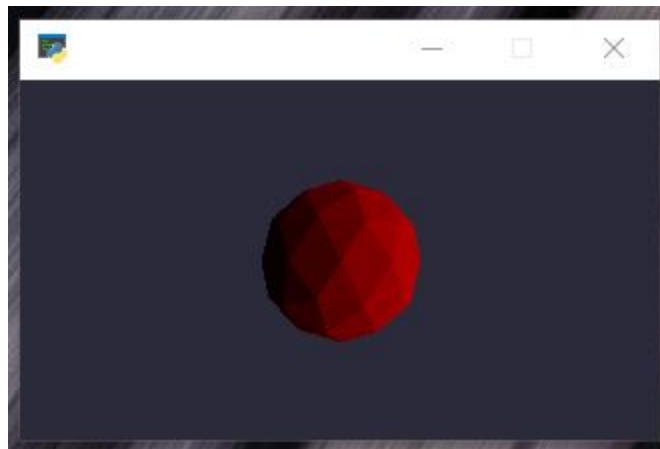
Once the renderer is started the Tkinter GUI will close and the SDL2 window will open, rendering the image. The SDL2 window cannot be interacted with, apart from being dragged around the screen, and closed.



The main window, for customising camera-specific variables



The model window, for customising variables specific to a single model



The render window, showing the finished image after rendering

Hardware

I will compile the renderer for use on a x64 desktop running Windows 10, and the GUI will be launched using the official Python interpreter, as that is what I have.

The only dependencies for compilation of the renderer will be a C++ compiler and a copy of the SDL2 library. Both C++ and SDL2 enjoy support for a large variety of hardware and operating systems, from Linux and MacOS to the Raspberry Pi and Nintendo Switch.

For Python, there are plenty of tools that take an existing Python script and allow it to be run on many different devices.

Explicit hardware acceleration like multithreading or GPUs will not be used, so it will be able to run even on a single core device.

So overall, the program will need minimal changes to run on any device that could be needed.

Data Structures

OBJ files

An OBJ file stores the data of a 3D model in secondary storage.

Each line of a OBJ file defines a different part of the model. The character (or characters) at the start of the line define what type of data is stored, and the rest of the line stores the data itself.

Vertices ('v') and vertex normals ('vn') are stored as a sequence of 3 floating point numbers, in order representing the x, y, and z components of its 3 dimensional vector.

Triangles ('f') are stored as a sequence of three strings of the form 'v//n' where 'v' is the integer index of the vertex and 'n' is the integer index of the normal for that vertex. Indices are by the order of vertices or normals in the file, and start at 1. These three strings give the three vertices (and a vertex normal for each) of the triangle. The order of these strings matters, as blender exports them in anticlockwise winding order. Swapping two of these strings will reverse the facing of the triangle.

I will use OBJ files instead of another format because of its simplicity, as it can be tailored in Blender to only include the data needed. For example I will not be using texture mapping, so including texture coordinates in the files is unnecessary. Blender allows me to exclude them from the file output.

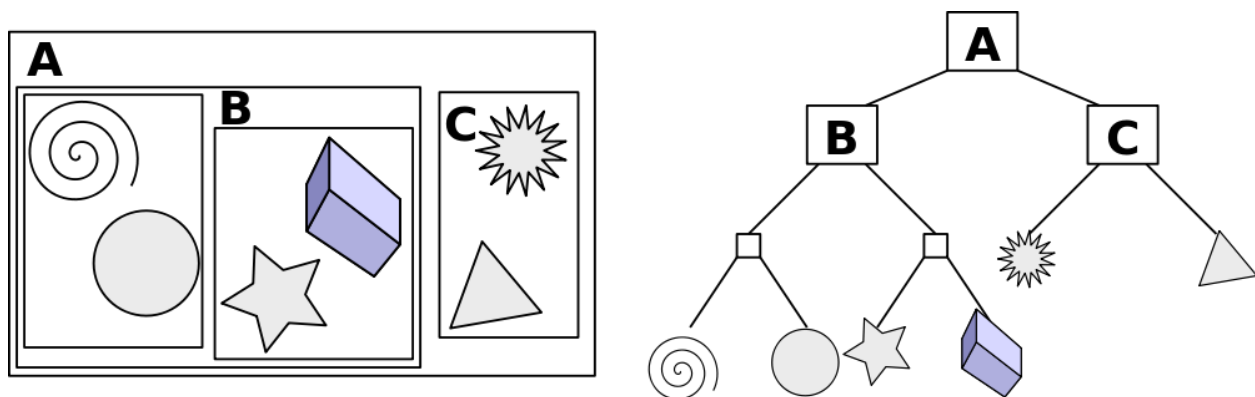
The data is also provided in an indexed format to reduce the redundancy caused by each triangle storing its vertices and normals raw, as there is lots of overlap (it being a closed triangle

mesh and all). This indexing saves a lot of memory at the computational cost of a small amount of pointer indirection.

Contrasting with other formats, OBJ stores its data in a human readable ASCII format, rather than binary. This makes debugging much easier.

I will be exporting OBJ files from the 3D modelling application 'Blender', which allows a degree of customisation in how the files are exported. This allows the inclusion of UV coordinates or interpolated normals for example, if my program was to be extended to utilise them.

Bounding volume hierarchy (BVH)²



The set of all triangles in the mesh of a model can be spatially partitioned into subsets, and bounding volumes created around these subsets that are more efficient to test for collisions against than testing against every element in the subset individually, with no risk of false negatives, only false positives which can then be verified as true or false.

Each of the subsets can then themselves be spatially partitioned and have bounding volumes created for them. This happens recursively until the number of elements in a subset is below some maximum, most likely the point where iterating over all the elements is faster than further testing against recursive bounding volumes. This value will probably be found experimentally and then hard coded. These bounding volumes containing the triangles with no children are called leaf nodes.

A ray that wants to test for collisions with the model first tests for a collision with the root bounding box. If there is no collision, then we know there can't possibly be a collision with any of the geometry and so we don't have to do anymore checks. If there is a collision, we then

²

http://www-ljk.imag.fr/Publications/Basilic/com.lmc.publi.PUBLI_Inproceedings@117681e94b6_1860ffd/bounding_volume_hierarchies.pdf

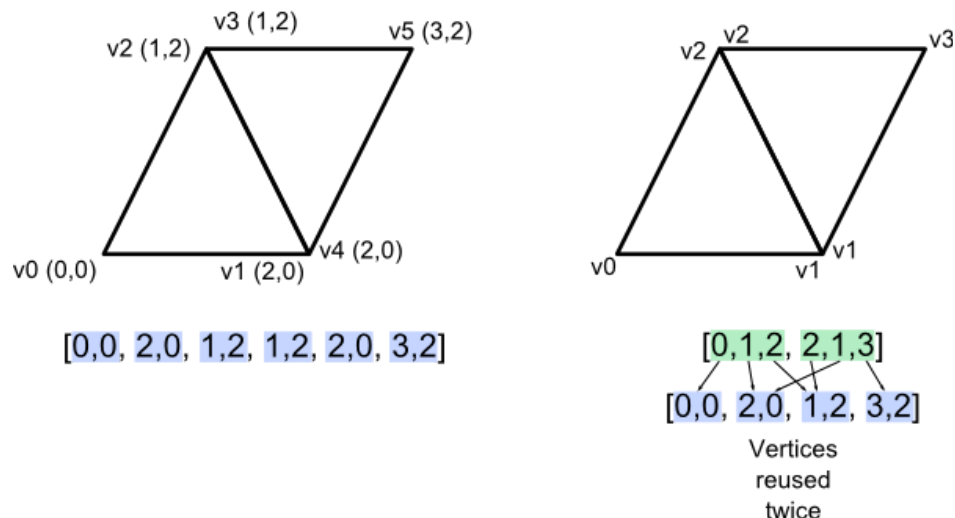
check for collisions with the root volume's children. This happens recursively until we successfully collide with a leaf node, at which point we test for collisions with each of the triangles individually.

This is far more efficient than checking for collisions with every single triangle, as large subsets of the triangles can be discarded with a simple check. In terms of computational complexity, checking against every triangle is $O(n)$, while checking against a BVH is $O(\log_2 n)$, where n is the number of triangles. For even a small n the BVH is much faster.

While construction of the BVH takes time, of $O(n \log_2 n)$ time complexity, the gains in speed during the ray casting makes this insignificant, as the ray collisions are calculated for every ray, of which there will be potentially millions, while the BVH construction is done once.

To derive the construction time of the BVH, consider that for each recursion every triangle must be iterated over once, for n operations, and that as the sets are being split into two subsets, the number of recursions will be, on average, $\log_2(n - m + 1)$, where m is the maximum number of triangles per leaf node. Thus the total running time will be $n \log_2(n - m + 1) = O(n \log_2 n)$, as m is a constant value, and adding a constant to an exponent doesn't change the complexity class³.

Indexed vertices



Storing vertex indices in a triangle instead of the raw vertices allows the redundancy to be eliminated, and the memory used to be reduced greatly. If a vertex is shared by n edges, then the memory saved is $\frac{n-1}{n}$, as a fraction of the original memory footprint. While this doesn't

³ <https://www.d.umn.edu/~ddunham/cs3512s10/notes/l12.pdf>

include the memory taken up by the index, compared to 3D vectors it's pretty much insignificant.

However there is a performance cost in using this pointer indirection. A flat contiguous list of vertices, while taking a significantly larger amount of memory, would be able to be stored in a sequence of cache lines, making their referencing much faster than storage in main memory, which is used with the pointer indirection.

This performance cost is acceptable, especially as a flat list would be difficult to work with when constructing a BVH.

Vectors and matrices

I will only be using 3D vectors, and 3 by 3 matrices, as I am working in 3D space.

A vectors is represented as three single precision floating point numbers, the x, y, and z components.

```
struct Vec3 {  
    float x, y, z;  
};
```

A matrix is represented as nine single precision floating point numbers, each representing a separate component. While technically the memory layout of the matrices is in column-major order, the matrix components are indexed with unique variable names rather than indices, so it has no effect in use.

```
struct Mat3 {  
    float x0, y0, z0, x1, y1, z1, x2, y2, z2;  
};
```

Vectors and matrices have been kept without extra bookkeeping data to minimise the memory used, as the program creates many thousands of these objects, vectors especially.

I am using single precision floating point numbers because the program was much faster than with double precision floats. Testing is included in it's own section, but in essence, most of the time spent working with vectors and matrices is in the initialisation of new objects. Using single precision floats means half the memory footprint, and so reduces the time taken to initialise the objects and move them around in memory.

The components are implicitly defined as public (the default class access modifier for C++ 'structs') as there's no reason to hide the data. Defining distinct get and set functions for each component is completely unnecessary, as these structures aren't doing any processing that

needs to be hidden. It would also be extremely verbose, especially for 'Mat3' with it's nine separate components.

Rays

Rays are defined as an origin vector O , or 'anchor point', with a direction vector D . This defines an analytic representation of a line,

$$P = O + tD$$

which can be treated as a half-line as it encodes direction, in the sign of the parameter t . This sign allows the program to detect if intersections happen with geometry behind the camera, and ignore them accordingly.

```
struct Ray {  
    private:  
        Vec3 origin;  
        Vec3 direction;
```

Models

Models store the data of a 3D mesh to be rendered. They are defined by a center, as well as ordered lists of vertices and normals indexed by a collection of triangles.

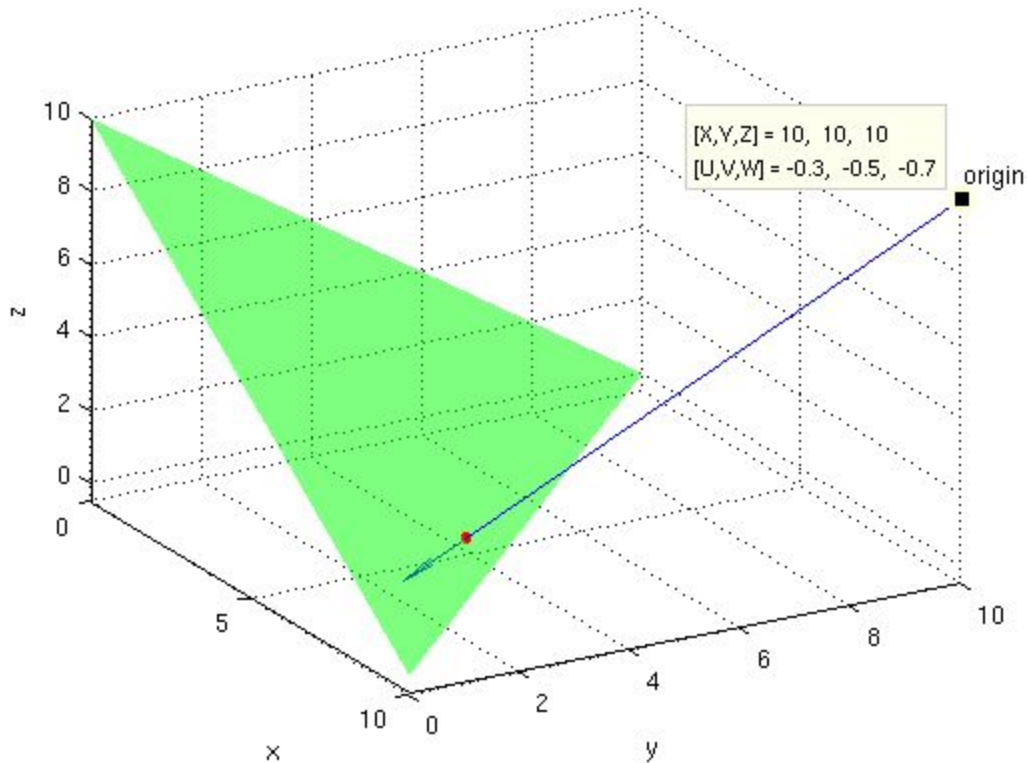
```
struct Model {  
    private:  
        Vec3 position;  
        std::vector<Triangle> triangles;  
        std::vector<Vec3> vertices;  
        std::vector<Vec3> normals;  
        std::shared_ptr<BVHNode> rootNode;
```

'rootNode' is a pointer to the root node of the BVH associated with the model. It is a shared pointer as other pointers to this root node exist in other places throughout the program.

The triangles in the list 'triangles' store only indices to vertices and normals, as described in the 'indexed vertices' section. These indices are used to access the vertices and normals stored by the model in the lists 'vertices' and 'normals' respectively.

Algorithms

Ray-triangle intersection⁴



I will use the Möller–Trumbore intersection algorithm for finding collisions between rays and triangles. It will return both whether there was an intersection, and the distance to it from the ray origin.

The barycentric coordinates of a triangle are 3 real coefficients w, u, v each scaling one of the vertices of the triangle, A, B, C , giving some point on the triangle P .

$$P = wA + uB + vC$$

With the restrictions that,

$$w, u, v \geq 0$$

4

<https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection>

$$w + u + v = 1$$

The intersection point of the ray and the triangle is parameterised in terms of barycentric coordinates.

$$P = wA + uB + vC$$

The formula can be rearranged as so,

$$P = (1 - u - v)A + uB + vC$$

$$P = A + u(B - A) + v(C - A)$$

The intersection point is also represented by the ray's parametric equation.

$$P = O + tD$$

Where t is the distance of the ray from its origin O , and D is the normalised direction vector.

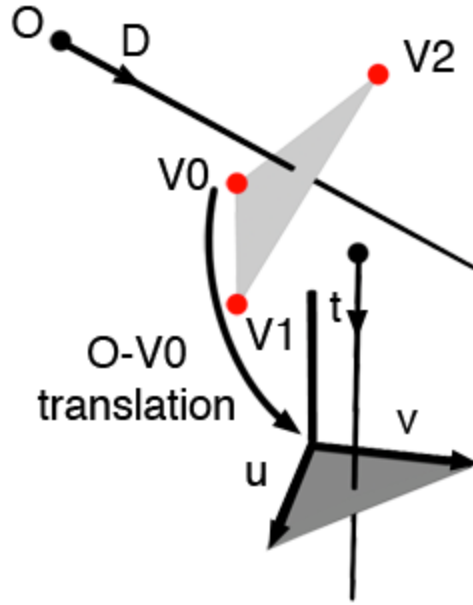
Rearranging:

$$O + tD = A + u(B - A) + v(C - A)$$

$$O - A = -tD + u(B - A) + v(C - A)$$

Which can be written in vector notation as:

$$\begin{bmatrix} -D \\ B-A \\ C-A \end{bmatrix} \bullet \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O-A$$



This essentially expresses the intersection point in ' t, u, v ' space, as opposed to ' x, y, z ' space. t gives the distance from the ray origin to the intersection point, and u, v give us the barycentric coordinates of the point with respect to the triangle.

We can now solve this linear equation for t, u, v using Cramer's Rule, which finds the solutions in terms of the determinants.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\begin{vmatrix} -D & B-A & C-A \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} O-A & B-A & C-A \end{vmatrix} \\ \begin{vmatrix} -D & O-A & C-A \end{vmatrix} \\ \begin{vmatrix} -D & B-A & O-A \end{vmatrix} \end{bmatrix}$$

If both u, v are within the range $[0, 1]$ and $u + v \leq 1$, then the intersection point is within the triangle, and t is the valid distance to the intersection point from the ray's origin.

If either of these conditions are false, then the calculated intersection point is outside of the triangle, and t is not valid.

While u, v can be used for things like texture mapping etc. I will only use them for verifying that the intersection point is valid.

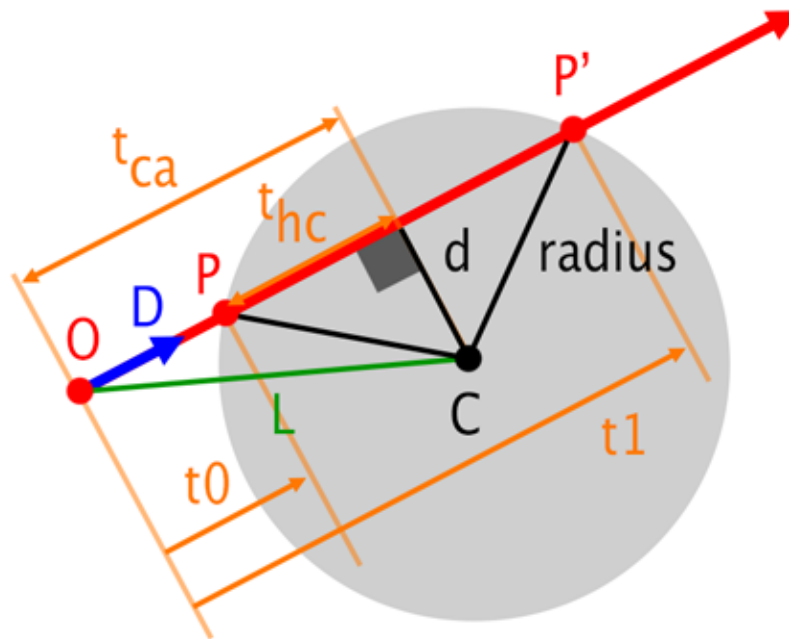
Back-face culling can also be easily added to the algorithm by comparing the ray direction with the winding order of the triangle. This increases performance by discarding unnecessary triangles, as the back-faces should never be seen normally, given a closed mesh. It also makes it

visually very obvious when the winding order is the wrong way round, as the model appears 'inside out'.

Pseudocode:

```
FUNCTION rayTriangleIntersection(  
    Vec3 orig, Vec3 dir,  
    Vec3 v0, Vec3 v1, Vec3 v2,  
    reference to float u,  
    reference to float v,  
    reference to float t)  
Vec3 v0v1 = v1 - v0          // Vectors representing edges of  
Vec3 v0v2 = v2 - v0          // the triangle  
Vec3 pvec = dir.cross(v0v2)  
float det = v0v1.dot(pvec)    // If the determinant is positive  
IF det < kEpsilon THEN       // it is a front facing triangle  
    RETURN FALSE             // An epsilon value is used  
ENDIF                         // instead of zero as there may be  
float invDet = 1 / det;       // floating point precision errors  
Vec3 tvec = orig - v0  
u = tvec.dot(pvec) * invDet    // Calculate u value  
IF u < 0 OR u > 1 THEN        // If u is outside of the bounds  
    RETURN FALSE             // of the triangle return false  
ENDIF  
Vec3 qvec = tvec.cross(v0v1)  
v = dir.dot(qvec) * invDet     // Calculate v value  
IF v < 0 OR u + v > 1 THEN    // If v is outside of the bounds  
    RETURN FALSE             // of the triangle return false  
ENDIF  
t = v0v2.dot(qvec) * invDet    // Calculate t value  
RETURN TRUE                   // Return successful collision  
ENDFUNCTION
```

Ray-sphere intersection⁵



Spheres will be used as the bounding volumes for the BVHs due to their simplicity and the efficiency of their ray intersection algorithm.

I will solve to find the intersection distance with a geometric solution, in which the distance to the two intersection points is calculated, and the distance to the closest intersection point is returned through an out parameter.

This is done instead of returning the actual intersection point as the distance is more useful (for example pixel occlusion), and even if it's needed, it's easier to calculate the intersection point from the distance, by simply substituting into the ray equation, than to calculate the distance from the intersection point by the pythagorean theorem, which includes a square root.

Pseudocode:

```
FUNCTION raySphereIntersect(  
  Vec3 orig, Vec3 dir,  
  Vec3 center, float radius,  
  float& t)  
  float t0, t1 // Distances to intersections  
  Vec3 l = center - orig  
  float tca = l.dot(dir) // Projected distance
```

5

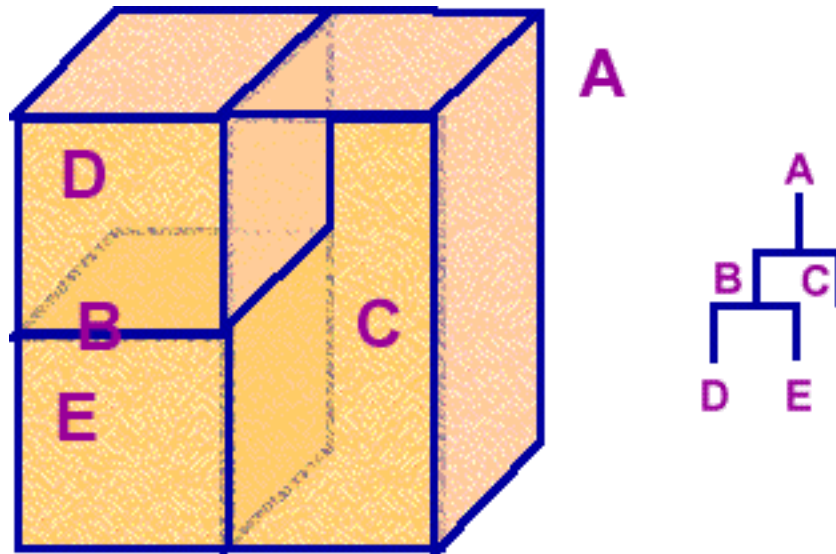
<https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection>


```

float d2 = l.dot(l) - tca * tca           // to the center
IF d2 > radius * radius THEN             // Compare distance of closest
    RETURN FALSE                         // approach to radius
ENDIF
float thc = sqrt(radius * radius - d2) // Projected distance from
t0 = tca - thc                           // center to intersection
t1 = tca + thc                           // points
IF t0 > t1 THEN
    swap(t0, t1)                         // Order intersection
ENDIF                                    // points by distance
IF t0 < 0 THEN
    t0 = t1
if (t0 < 0) RETURN FALSE
ENDIF
t = t0                                   // Set out variable to closest
RETURN TRUE                             // intersection point
ENDFUNCTION

```

Partition triangles



At each level of the bounding volume hierarchy, the set S of triangles must be spatially partitioned into two subsets, S_1 and S_2 . My algorithm splits S by finding optimally placed, axis-aligned planes, constructed to split S into subsets that are both as equally sized and spatially distinct as possible.

The subsets must be kept equally sized so that the traversal time of the hierarchy is kept even across the whole model, and minimising the worst case traversal time.

The subsets must be as separated as possible so that the overlap of their bounding volumes is minimised, thus improving the overall efficiency of ray intersection calculations.

First, to find the optimal axis by which to split S , the variance of each component of the triangle's centers are found, i.e. the variance in the x-axis is found, then the y-axis, then the z-axis. The center of a triangle is cached in it's data structure, as the partitioning algorithm runs many times.

Once the optimal axis is found, the triangles are iterated over again, to split them into the two subsets. The relevant component of centers of the triangles are compared to the mean value, and split depending on whether they are less than or greater than this mean.

Each subset is then passed to a child BVH node, where the process starts again, recursively.

In parallel with this, the bounding volume of S is also being calculated. The center of the sphere is the average of all the centers of the triangles, and the radius is the maximum distance of any of the vertices from this center, so that the entirety of S is contained within this bounding sphere.

Pseudocode:

```
FUNCTION calcAxisWithGreatestVariance()
Vec3 mean, sumOfSqrs
FOR int i FROM 1 TO triangles.size() REPEAT // Iterate over triangles
    Vec3 center = triangles[i].getCenter()
    Vec3 oldMean = mean
    mean = mean + (center - mean) / i           // Iterative mean
    sumOfSqrs = sumOfSqrs                       // Iterative squares sum
                + (center - mean)
                * (center - oldMean)
ENDFOR
RETURN getAxis(sumOfSqrs)
ENDFUNCTION
```

'getAxis' returns an enum representation of the axis with the greatest variance.

```
namespace Axes {
enum Axes {
    x, y, z
};
}
```

I am using Welford's method⁶ for computing the variance, as it means it can be calculated in one pass of the list of triangles rather than two, which the standard formula requires.

$$s^2 = \frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N-1}$$

s^2 is the variance of the data (the square of the standard deviation), N is the number of triangles in the subset. x_i is the i -th triangle in the subset, as in the formula the sigma symbol (Σ) means that for each element x_i of the subset, $(x_i - \bar{x})^2$ is calculated and then they are all summed together.

The use of the mean \bar{x} within the loop means that two passes are needed, as \bar{x} requires a single pass by itself and can't be calculated in parallel with the main pass.

While this formula can be rearranged to a single pass version, it is very numerically unstable, occasionally giving results with large relative error and sometimes even negative variances.

Rearranging,

$$s^2 = \frac{\sum_{i=1}^N (x_i^2 - 2x_i\bar{x} + \bar{x}^2)}{N-1} = \frac{\sum_{i=1}^N x_i^2 - 2N\bar{x} + N\bar{x}^2}{N-1} = \frac{\sum_{i=1}^N x_i^2 - N\bar{x}^2}{N-1}$$

Welford's method,

$$s^2 = \frac{\sum_{i=1}^N (x_i - \bar{x}_i)(x_i - \bar{x}_{i-1})}{N-1}$$

Where \bar{x}_i is the mean of all the values up to the i th element, and $\bar{x}_0 = 0$.

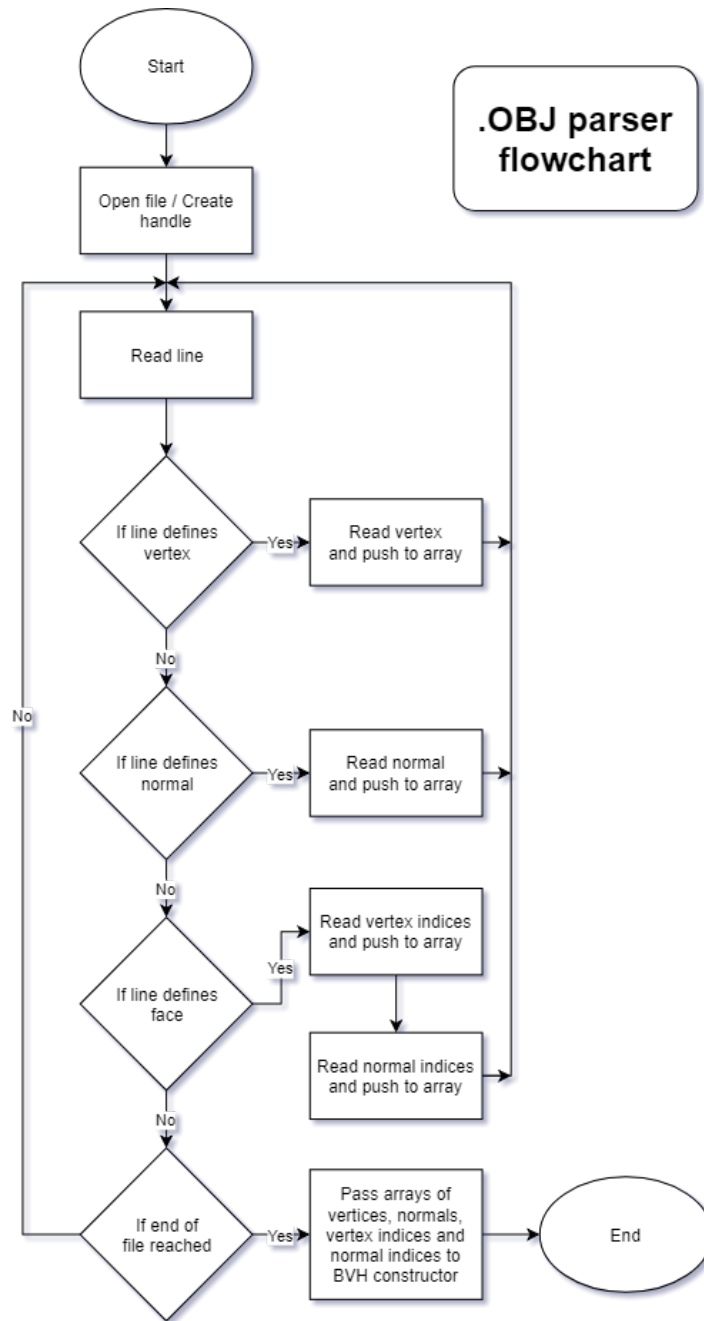
In my implementation I do not normalise the variance by dividing by $N - 1$, as this would be the number of triangles minus one, which is constant among all three axes. Dividing all three components by the same scalar has no effect on their relative size, which is all I'm using them for, so the normalisation is unnecessary.

OBJ file parser⁷

The file parser reads model data from an OBJ file, identifying vertices, normals, and faces (as indices to the vertex and normal lists), and filtering them accordingly into the required lists.

⁶ <http://jonisalonen.com/2013/deriving-welfords-method-for-computing-variance/>

⁷ <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/>

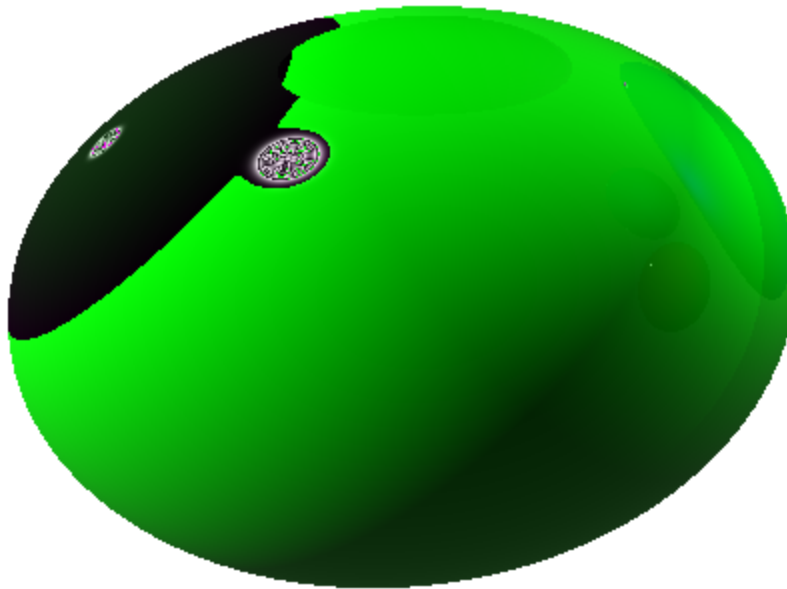


Reinhard tone-mapping operator⁸

During the rendering process, colours are represented in a high dynamic range (HDR) format, where brightness values range from zero (i.e. complete darkness) to infinity, as there is no maximum brightness. While this is a far more natural representation than a low dynamic range (LDR) format, the screen can only show colours within a finite range, and so using these values

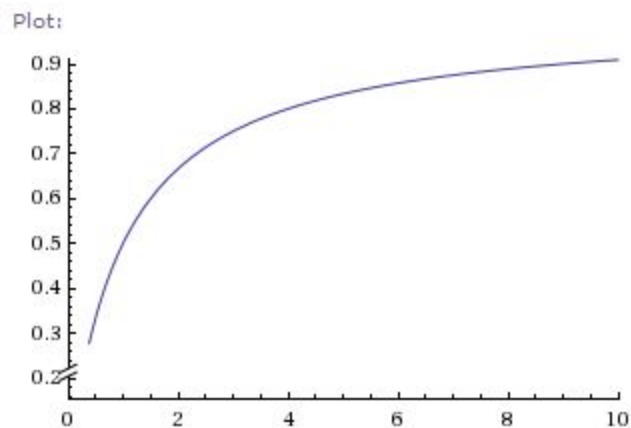
⁸<http://www.cs.utah.edu/~reinhard/cdrom/tonemap.pdf>

directly, at best, leads to flat ‘hotspots’ of colour where the values extend above the upper limit. At worst it causes undefined behaviour.



Tone-mapping maps these unbounded values to the $[0, 1]$ range, that can be represented on a screen. The Reinhard operator is one of the simplest tone-mappers.

$$L(x) = \frac{x}{1+x}$$



This takes a brightness value $x \in \mathbb{R}^+$, and applies the function L , which restricts the brightness value to the range $L(x) \in [0, 1]$, so that it can be represented on the screen.

This is preferable to a simple clamp to the maximum, as large values are smoothly compressed to the required range, rather than giving strong colour banding.

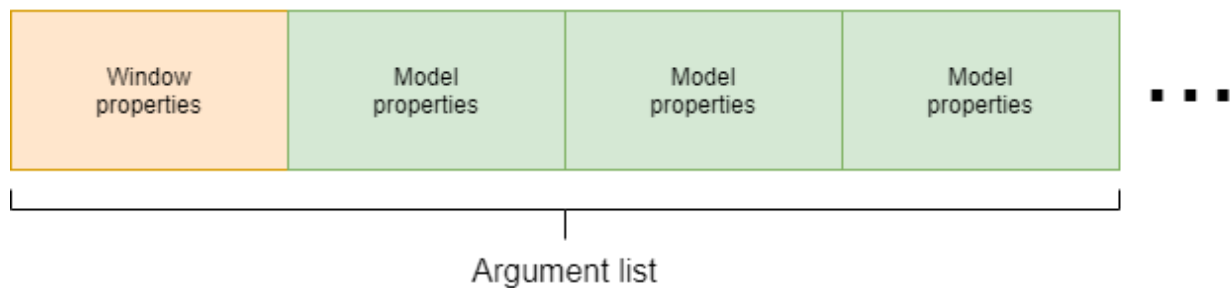
Sending data from Python to C++

C++ receives a tuple of arguments from Python, which contains raw pointers to native Python objects.

Python

The 'MainWindow' is responsible for argument packing, as well as initialising the renderer. It creates a mutable list of window-specific properties, such as window dimensions and camera position. It then iterates through the models and creates argument lists for them as well, each time the 'MainWindow' appending the model's list to its own.

This creates a list with a header of window properties, and then a tail of repeating model properties



Once completed, the mutable list is casted to a tuple, and passed as an argument to the rendering entry function.

Data is checked for validation before any packing occurs, so they don't need to be validated again before sending.

C++

The entry function receives two pointers to Python objects (of type 'PyObject*'), one to the module object, and one to the tuple of arguments being passed to the function from the GUI.

Each element of the tuple is also of type 'PyObject*', and must be casted to its required C++ type before being used. As validation and type checking was done by the GUI before sending the data, the C++ program assumes validity and does almost no checking of its own. One exception is that the number of arguments is checked to find whether it is consistent with the expected number of arguments.

Classes

Camera

Field name	Access modifier	Type	Description
<code>position</code>	Private	Vector 3	The camera's position in the scene
<code>aspectRatio</code>	Private	Float	The ratio of height to width of the screen
<code>horizontalFOV</code>	Private	Float	Horizontal field of view in degrees
<code>verticalFOV</code>	Private	Float	Vertical field of view in degrees
<code>pixelWidth</code>	Private	Integer	Screen width in pixels
<code>pixelHeight</code>	Private	Integer	Screen height in pixels
<code>halfPixelWidth</code>	Private	Integer	Half the screen width in pixels
<code>halfPixelHeight</code>	Private	Integer	Half the screen height in pixels
<code>distToProjPlane</code>	Private	Float	Perpendicular distance from the camera to its projection plane
<code>background</code>	Private	Vector 3	The colour of the background of the scene, to be used when there is no ray intersection with geometry
<code>lastModelIndex</code>	Private	Integer	Index of the last model in the 'models' vector, caching this is faster than reading the size of the vector

models	Private	List of shared pointer to Model	Stores pointers to the models in the scene
--------	---------	---------------------------------	--

Function name	Access modifier	Return type	Out parameters	Description
renderPixel	Private	None	None	Renders a single pixel
emitScreenRay	Private	Ray	None	Generates a ray according to the screen space position, and the distance to the projection plane.
getRayIntersectionColour	Private	Vector 3	None	Intersect the ray with the geometry, and get the colour from that
getCollisionIndices	Private	None	Integer modelIndex Integer triangleIndex	Get the index of the triangle and model that the ray intersects with most closely to the ray origin.
getBrightnessAtPoint	Private	Float	Integer modelIndex Integer triangleIndex	Get the brightness of a point in space
getBrightnessAtNormal	Private	Float	None	Get the brightness of a triangle at the normal
setDrawColour	Private	None	None	Using the SDL library, set the draw colour
toneMap	Private	Float	None	Convert HDR

				colours to LDR colours using a Reinhard operator
renderImage	Public	None	None	Renders the entire image
insertModel	Public	None	None	Insert a model's pointer into the camera's internal list

BVHNode

Field name	Access modifier	Type	Description
maxTriangleNumPerLeaf	Private	Integer	The maximum number of triangles to be stored in each leaf node of the BVH
child0	Private	Unique pointer to BVHNode	Pointer to the first child nodes in the BVH binary tree
child1	Private	Unique pointer to BVHNode	Pointer to the second child nodes in the BVH binary tree
center	Private	Vector 3	Average of all the centers of the triangles in the current set, this is cached as it's used multiple times in the rendering stage as the center of the bounding sphere
modelOffset	Private	Vector 3	The position of the model being stored
radius	Private	Float	Radius of the bounding sphere
isLeaf	Private	Boolean	Whether the node is

			a leaf or not
--	--	--	---------------

Function name	Access modifier	Return type	Out parameters	Description
setBoundRadius	Private	None	None	Set the radius of the bounding sphere of the node
calcBounds	Private	None	None	Calculate the radius of the bounding sphere of the node, to encompass all triangles in it's set
calcAxisWithGreatestVariance	Private	Axes	None	Calculate the axis with which the triangle's positions give the greatest variance
partitionX	Private	None	None	Partition the triangles into two subsets by the x-axis
partitionY	Private	None	None	Partition the triangles into two subsets by the y-axis
partitionZ	Private	None	None	Partition the triangles into two subsets by the z-axis
partition	Private	None	None	Partition the triangles into two subsets
raySphereIntersection	Private	Boolean	None	Find whether the ray intersects with

				the sphere
rayTrianglesIntersection	Private	Boolean	Float t Integer triangleIndex	Find the distance to the closest intersection point of the ray and the triangles in the node's set
recurseRayIntersection	Private	Boolean	Float t Integer triangleIndex	Intersect the ray with the node's children
build	Public	None	None	Build the tree, partitioning the triangles into two subsets and passing one to each child node
rayIntersection	Public	Boolean	Float t Integer triangleIndex	Find the intersection of a ray with the node

Model

Field name	Access modifier	Type	Description
position	Private	Vector 3	The position of the model
triangles	Private	List of Triangle	The triangles in the mesh of the model
vertices	Private	List of vector 3	The vertices of triangles in the mesh of the model. Triangles store indices to this list
normals	Private	List of vector 3	The normals of

			triangles in the mesh of the model. Triangles store indices to this list
rootNode	Private	Shared pointer to BVHNode	A pointer to the BVH used by the model

Function name	Access modifier	Return type	Out parameters	Description
getTriangle	Public	Triangle	None	Get a triangle, using an index to the internal list
getPosition	Public	Vector 3	None	Get the position of the model
getVertex	Public	Vector 3	None	Get a vertex, using an index to the internal list
getVertexNum	Public	Integer	None	Get the number of vertices in the internal list
getNormal	Public	Vector 3	None	Get a normal, using an index to the internal list
rayIntersection	Public	Boolean	Float t Integer triangleIndex	Intersect a ray with the model, using the internal BVH

Triangle

Field name	Access modifier	Type	Description
v0Index	Private	Integer	Index of the first vertex of the triangle in the BVHNode

v1Index	Private	Integer	Index of the first vertex of the triangle in the BVHNode
v2Index	Private	Integer	Index of the first vertex of the triangle in the BVHNode
normalIndex	Private	Integer	Index of the normal of the triangle in the BVHNode
triangleIndex	Private	Integer	Index of itself in the BVHNode
parent	Private	Shared pointer to Model	The model that this triangle belongs too. Used to access vertices and normals
planeOffset	Private	Float	The constant term in the equation of the plane of the triangle ⁹
center	Private	Vector 3	The mean average of the vertices of the triangle, this is cached as it's used multiple times during the rendering stage

Function name	Access modifier	Return type	Out parameters	Description
rayIntersection	Public	Boolean	Float t	Find the distance to the intersection point of the ray and the triangle
setParent	Public	None	None	Set the parent model of the triangle
getv0Index	Public	Integer	None	Get the index of the first vertex

⁹ <http://mathworld.wolfram.com/Plane.html>

				of the triangle
getv1Index	Public	Integer	None	Get the index of the second vertex of the triangle
getv2Index	Public	Integer	None	Get the index of the third vertex of the triangle
getNormalIndex	Public	Integer	None	Get the index of the normal of the triangle
getTriangleIndex	Public	Integer	None	Get the index of the triangle
getCenter	Public	Vector 3	None	Get the center of the triangle (i.e the average of the three vertices)

Ray

Field name	Access modifier	Type	Description
origin	Private	Vector 3	The anchor point of the ray, where it originates from
direction	Private	Vector 3	The direction that the ray 'travels'

Function name	Access modifier	Return type	Description
getOrigin	Public	Vector 3	Get the origin of the ray
getDirection	Public	Vector 3	Get the direction of the ray
setOrigin	Public	None	Set the origin of the ray

setDirection	Public	None	Set the direction of the ray
normalise	Public	None	Normalise the direction vector of the ray
project	Public	Vector 3	Find the point at distance 't' from the ray origin along the direction vector

Vec3

Field name(s)	Access modifier	Type	Description
x, y, z	Public	Float	The X, Y, and Z component of the vector

Function name	Access modifier	Return type	Description
dot	Public	Float	Dot product of the two vectors
cross	Public	Vector 3	Cross product of the two vectors
Addition override (vector-vector)	Public	Vector 3	Component-wise addition of the two vectors
Subtraction override (vector-vector)	Public	Vector 3	Component-wise subtraction of the two vectors
Multiplication override (vector-scalar)	Public	Vector 3	Multiply the vector by a scalar
Multiplication override (vector-vector)	Public	Vector 3	Component-wise multiplication of the two vectors
Division	Public	Vector 3	Divide the vector by a

override (vector-scalar)			scalar
getLength	Public	Float	Get the length of the vector (by the pythagorean theorem)
normalise	Public	Vector 3	Get a normalised version of the vector (i.e of unit magnitude)
getReverse	Public	Vector 3	Get the negative of the vector, i.e facing in the reverse direction with equal magnitude

Mat3

Field name(s)	Access modifier	Type	Description
x0, y0, z0, x1, y1, z1, x2, y2, z2	Public	Float	The components of the matrix, in column-major order

Function name	Access modifier	Return type	Description
getSquare	Public	Matrix 3x3	Multiply the matrix by itself
Addition override (matrix-matrix)	Public	Matrix 3x3	Component-wise addition of the two matrices
Multiplication override (matrix-scalar)	Public	Matrix 3x3	Multiply the matrix by a scalar
Multiplication override (matrix-vector)	Public	Vector 3	Multiply a vector by the matrix
Multiplication	Public	Matrix 3x3	Multiply the two

<code>override (matrix-matrix)</code>			matrices
---	--	--	----------

Non-class functionality

ModelLoader

Function name	Return type	Out parameters	Description
<code>loadOBJ</code>	Boolean	List of Vector 3 <code>outVertices</code> List of Vector 3 <code>outNormals</code> List of integer <code>outVertexIndices</code> List of integer <code>outNormalIndices</code>	Opens OBJ file and reads data into lists
<code>parseVertex</code>	None	List of Vector 3 <code>tempVertices</code>	Read vertex from the file and add to the vertex list
<code>parseNormal</code>	None	List of Vector 3 <code>tempNormals</code>	Read normal from the file and add to the normal list
<code>parseFace</code>	None	List of integer <code>vertexIndices</code> List of integer <code>normalIndices</code>	Read vertex and normal indices from the file and add to their respective lists
<code>parseOBJ</code>	None	List of integer <code>vertexIndices</code> List of integer <code>normalIndices</code> List of Vector 3 <code>tempVertices</code>	Read data from the file line by line, filtering the data into the required lists

		List of Vector 3 tempNormals	
--	--	---------------------------------	--

GUI classes

MainWindow

Field name	Type	Description
modelList	List of ModelData	A list of the models, shown visually in the model listbox
validArguments	Boolean	Describes whether the data in the entries is valid or not
camxEntry	Tkinter Entry	Entry for camera x position
camyEntry	Tkinter Entry	Entry for camera y position
camzEntry	Tkinter Entry	Entry for camera z position
camFOVEntry	Tkinter Entry	Entry for camera field of view
widthEntry	Tkinter Entry	Entry for window width
heightEntry	Tkinter Entry	Entry for window height
listbox	Tkinter Listbox	Listbox for showing the models in the scene
startButton	Tkinter Button	Button for starting the renderer
exitButton	Tkinter Button	Button for exiting the GUI
addButton	Tkinter Button	Button for adding a model to the listbox
deleteButton	Tkinter Button	Button for deleting the selected model from the listbox

Function name	Return type	Description
configureWindow	None	Sets all the required window

		properties
setBinds	None	Binds keyboard inputs to their required functions
initLabels	None	Initialise labels in the main window, and place them
initEntries	None	Initialise all entries
initListboxes	None	Initialise listboxes, and bind keyboard inputs to the required functions
initButtons	None	Initialise buttons and bind keyboard inputs to the required functions
start	None	Validate inputs, and if valid, start the renderer
validate	None	Validate data in the entries, and if invalid, highlighting the entries red and preventing renderer from starting
createAttrbValidityDict	Dictionary	Creates a dictionary where keys are the attributes to be validated, and the values are boolean values describing whether the given attribute is valid
getValidation	Dictionary	Check for validity for each entry, setting values in the validity dictionary accordingly
validFloat	Boolean	Validate whether a string could be converted to a valid floating point number
highlightInvalidEntries	None	Go through the entries, if the input is valid set the background to white, if invalid set it to red
spawnModelWindow	None	Create a model window for the selected model to that the user can change variables

		for it
<code>quitGUI</code>	None	Destroy the window, and exit the Python application
<code>deleteModel</code>	None	Delete the selected model from the model listbox
<code>getDefaultModel</code>	ModelData	When a user adds a model to the listbox, it must have some default data to start with, this gives the default configuration
<code>addModel</code>	None	Adds the default model to the listbox, and opens a model window for it so that the user can configure it
<code>updateModelName</code>	None	Update the listed name for a given model in the listbox to it's latest name
<code>packArguments</code>	Tuple	Pack the window and model attributes into a tuple that can be passed to the C++ application
<code>initRender</code>	None	Start the renderer with the required arguments
<code>getWidth</code>	Integer	Get the value from the window width entry, converted from a string to an integer
<code>getHeight</code>	Integer	Get the value from the window height entry, converted from a string to an integer
<code>getCamx</code>	Float	Get the value from the camera x position entry, converted from a string to a float
<code>getCamy</code>	Float	Get the value from the camera y position entry,

		converted from a string to a float
getCamz	Float	Get the value from the camera z position entry, converted from a string to a float
getCamFOV	Float	Get the value from the camera field of view entry, converted from a string to a float

ModelWindow

Field name	Type	Description
parent	MainWindow	The main window responsible for this model window
model	ModelData	The model that is being configured
validModelArguments	Boolean	Describes whether the inputs to the entries are valid
filenameEntry	Tkinter Entry	Entry for the filename
xEntry	Tkinter Entry	Entry for the model x position
yEntry	Tkinter Entry	Entry for the model y position
zEntry	Tkinter Entry	Entry for the model z position
rotxEntry	Tkinter Entry	Entry for the model x rotation
rotyEntry	Tkinter Entry	Entry for the model y rotation
rotzEntry	Tkinter Entry	Entry for the model z rotation
flipxCheck	Tkinter Check Button	Check Button for the model mirroring in the x axis
flipyCheck	Tkinter Check Button	Check Button for the model mirroring in the x axis

<code>flipzCheck</code>	Tkinter Check Button	Check Button for the model mirroring in the x axis
<code>cancelButton</code>	Tkinter Button	Button for cancelling configuration of the model, cancelling all changes
<code>acceptButton</code>	Tkinter Button	Button for accepting configuration of the model, saving all changes

Function name	Return type	Description
<code>configureWindow</code>	None	Sets all the required window properties
<code>initLabels</code>	None	Initialise labels in the main window, and place them
<code>initEntries</code>	None	Initialise all entries
<code>initFilenameEntry</code>	None	Initialise filename entry
<code>initPosEntries</code>	None	Initialise position entries
<code>initRotEntries</code>	None	Initialise rotation entries
<code>initFlipChecks</code>	None	Initialise flip check buttons
<code>initWindowButtons</code>	None	Initialise buttons
<code>setBinds</code>	None	Binds keyboard inputs to their required functions
<code>windowExit</code>	None	Save model data, and if valid, exit the window
<code>saveAndValidate</code>	None	Save and validate the model data
<code>saveModelData</code>	None	Save the values in the entries to the model
<code>validateModelData</code>	None	Validate the data in the entries, highlight invalid entries, and if invalid entries exist, prevent the window

		from closing
highlightInvalidEntries	None	Highlight entries white if the data is valid, and red if invalid
highlightInvalidPosEntries	None	Highlight position entries as described above
highlightInvalidRotEntries	None	Highlight rotation entries as described above
destroyWindow	None	Destroy the model window, and return to the main window

ModelData

Field name(s)	Type	Description
filename	String	The name of the OBJ file that will be read from
x, y, z	Float	The x, y, and z coordinates of the model in world space
rotx, roty, rotz	Float	The rotation of the model, in degrees around the x, y, and z axes
flipx, flipy, flipz	Float	Whether the model will be mirrored along the x, y, and z axes
name	String	The name of the model, that will be used as it's label in the listbox in the main window

Function name	Return type	Description
createAttrbValidityDict	Dictionary	Creates a dictionary where keys are the attributes to be validated, and the values are boolean values describing whether the given attribute is valid

getValidation	Dictionary	Check for validity for each entry, setting values in the validity dictionary accordingly
validFilename	Boolean	Check whether the inputted filename is valid
validFloat	Boolean	Validate whether a string could be converted to a valid floating point number
packArguments	List	Pack the attributes of the model into a list

Data dictionaries

Python to C++ communication

Window specific data

Variable name	Type	Description
WIDTH	Integer	The width in pixels of the window
HEIGHT	Integer	The height in pixels of the window
camPos.x	Float	The x component of the position of the camera in world space
camPos.y	Float	The y component of the position of the camera in world space
camPos.z	Float	The z component of the position of the camera in world space
camFOV	Float	The field of view of the camera in degrees

Model specific data

Variable name	Type	Description
filename	String	The filename of the file containing the model data for the model
x	Float	The x component of the position of the model in world space
y	Float	The y component of the position of the model in world space
z	Float	The z component of the position of the model in world space
rotx	Float	The rotation of the model around the x axis in degrees
roty	Float	The rotation of the model around the y axis in degrees
rotz	Float	The rotation of the model around the z axis in degrees
flipx	Boolean	Whether the model should be mirrored in the x axis
flipy	Boolean	Whether the model should be mirrored in the yaxis
flipz	Boolean	Whether the model should be mirrored in the z axis

Model specific data is repeated for each model added to the scene

Technical solution

C++

module.cpp

```
// Disable warnings for unsafe file I/O
#define _CRT_SECURE_NO_WARNINGS

#include <Python.h>
#include <Windows.h>

#include <iostream>
#include <memory>
#include <chrono>
#include <string>
#include <SDL.h>
#include "camera.h"
// Forward declare entryPoint for use in the Python extension
void entryPoint(PyObject*, PyObject* args);
// Define the method(s) that are to be exported, and be publicly visible
static PyMethodDef rayTracerMethods[] = {
    {"render", (PyCFunction)entryPoint, METH_O, nullptr},
    {nullptr, nullptr, 0, nullptr}
};
// Define the information for the module itself
static PyModuleDef rayTracerModule = {
    PyModuleDef_HEAD_INIT,
    "RayTracer",
    "Renders images",
    0,
    rayTracerMethods
};
// Create the module
PyMODINIT_FUNC PyInit_RayTracer() {
    return PyModule_Create(&rayTracerModule);
}
// Print data to the Python console
static void print(std::string str) {
    PySys_WriteStdout(str.c_str());
    PySys_WriteStdout("\n");
}
```

```

// -----

// Screen dimensions
static int WIDTH;
static int HEIGHT;
// Camera field of view in degrees
static float camFOV;
// Camera position in world space
static Vec3 camPos;
// Root path for the OBJ files
static std::string root = "C:\\Users\\Mirrorworld\\Desktop\\NEA\\OBJ
files\\";

// Class containing SDL functionality
struct Context {
    SDL_Window* window;
    SDL_Renderer* renderer;
    bool initFailure;
};

// Class for intermediate storage of model information
struct ModelData {
    std::string filename;
    float x, y, z, rotx, roty, rotz;
    bool flipx, flipy, flipz;
};

static Context initialise() {
    // Seed random number generator
    srand((unsigned int)time(NULL));
    Context context;
    // Initialise SDL
    if (SDL_Init(SDL_INIT EVERYTHING) < 0) {
        context.initFailure = EXIT_FAILURE;
        return context;
    }
    // Initialise the SDL window and renderer
    SDL_CreateWindowAndRenderer(WIDTH, HEIGHT, SDL_WINDOW_ALLOW_HIGHDPI,
    &context.window, &context.renderer);
    if (context.window == NULL) {
        context.initFailure = EXIT_FAILURE;

```

```

        return context;
    }
    context.initFailure = EXIT_SUCCESS;
    return context;
}

static std::string pyObjectToString(PyObject* obj) {
    // Get the Python string 'repr' of the object
    PyObject* repr = PyObject_Repr(obj);
    // Convert to a Unicode Python string
    PyObject* pyStr = PyUnicode_AsEncodedString(repr, "utf-8", "String
decode failed");
    // Convert to a C++ string
    std::string str = std::string(PyBytes_AsString(pyStr));
    // Strip single quotes from the string, which are for some reason
included
    str.erase(std::remove(str.begin(), str.end(), '\\'), str.end());
    return str;
}

static ModelData parseModel(PyObject* args, int offset, int stride, int
num) {
    ModelData model = ModelData();
    // Calculate starting index of model data in the argument tuple
    const int start = offset + stride * num;
    // Get the filename, and cast from a PyObject to a std::string
    model.filename = pyObjectToString( PySequence_GetItem(args,
start));
    // Get PyObject floats, and cast them into C++ floats
    model.x = (float)PyFloat_AsDouble( PySequence_GetItem(args,
start + 1));
    model.y = (float)PyFloat_AsDouble( PySequence_GetItem(args,
start + 2));
    model.z = (float)PyFloat_AsDouble( PySequence_GetItem(args,
start + 3));
    model.rotx = (float)PyFloat_AsDouble( PySequence_GetItem(args,
start + 4));
    model.rotz = (float)PyFloat_AsDouble( PySequence_GetItem(args,
start + 5));
    model.rotz = (float)PyFloat_AsDouble( PySequence_GetItem(args,
start + 6));
    // Get PyObject integers, and cast to C++ ints, which then implicitly

```

cast to C++ booleans

```
    model.flipx = (int) PyNumber_AsSsize_t(PySequence_GetItem(args,
start + 7), NULL);
    model.flipy = (int) PyNumber_AsSsize_t(PySequence_GetItem(args,
start + 8), NULL);
    model.flipz = (int) PyNumber_AsSsize_t(PySequence_GetItem(args,
start + 9), NULL);
    return model;
}
```

```
static void parseNonModelAttrbs(PyObject* args) {
    WIDTH = (int) PyNumber_AsSsize_t(PySequence_GetItem(args, 0),
NULL);
    HEIGHT = (int) PyNumber_AsSsize_t(PySequence_GetItem(args, 1),
NULL);
    camPos.x = (float)PyFloat_AsDouble( PySequence_GetItem(args, 2));
    camPos.y = (float)PyFloat_AsDouble( PySequence_GetItem(args, 3));
    camPos.z = (float)PyFloat_AsDouble( PySequence_GetItem(args, 4));
    camFOV = (float)PyFloat_AsDouble( PySequence_GetItem(args, 5));
}
```

```
static bool parseArgs(PyObject* args, std::vector<ModelData>& models) {
    const int nonModelAttrbNum = 6;
    const int attrbPerModel = 10;
    const int argNum = (int)PyTuple_Size(args);
    // Find whether the number of arguments is consistent with the number
expected
    if ((argNum - nonModelAttrbNum) % attrbPerModel != 0) {
        return EXIT_FAILURE;
    }
    // Calculate the number of models
    const int modelNum = (argNum - nonModelAttrbNum) / attrbPerModel;
    // Parse attributes for each model individually
    for (int i = 0; i < modelNum; i++) {
        models.push_back(parseModel(args, nonModelAttrbNum,
attrbPerModel, i));
    }
    // Parse attributes not associated with any models
    parseNonModelAttrbs(args);
    return EXIT_SUCCESS;
}
```

```

static std::shared_ptr<Camera> initCam(std::vector<ModelData>& models) {
    // Create a camera
    std::shared_ptr<Camera> cam(new Camera(camPos, WIDTH, HEIGHT,
camFOV));
    // Create each model with it's required data and put into the camera
    for (int i = 0; i < models.size(); i++) {
        Transform transform = Transform(
            models[i].rotx,
            models[i].roty,
            models[i].rotz,
            models[i].flipx,
            models[i].flipy,
            models[i].flipz
        );
        Vec3 pos = Vec3(
            models[i].x,
            models[i].y,
            models[i].z
        );
        std::string path = root + models[i].filename;
        std::shared_ptr<Model> model = std::make_shared<Model>(path,
pos, transform);
        cam->insertModel(model);
    }
    return cam;
}

static float getTimeElapsed(std::chrono::steady_clock::time_point start) {
    auto end = std::chrono::high_resolution_clock::now();
    auto dur = end - start;
    auto ms =
std::chrono::duration_cast<std::chrono::milliseconds>(dur).count();
    return ms / 1000.0f;
}

static void mainLoop(Context context) {
    SDL_Event windowEvent;
    while (true) {
        if (SDL_PollEvent(&windowEvent)) {
            if (SDL_QUIT == windowEvent.type) {
                return;
            }
        }
    }
}

```

```

        }
        SDL_RenderPresent(context.renderer);
    }
}

static bool quit(Context context) {
    SDL_DestroyWindow(context.window);
    SDL_Quit();
    return EXIT_SUCCESS;
}

void entryPoint(PyObject*, PyObject* args) {
    std::vector<ModelData> models = std::vector<ModelData>();
    if (parseArgs(args, models) == EXIT_FAILURE) return;
    Context context = initialise();
    if (context.initFailure == EXIT_FAILURE) return;

    std::shared_ptr<Camera> cam = initCam(models);
    auto start = std::chrono::high_resolution_clock::now();
    cam->renderImage(context.renderer, WIDTH, HEIGHT);

    mainLoop(context);

    quit(context);
    return;
}

```

camera.h

```

#pragma once

#include <vector>
#include <SDL.h>
#include "model.h"

struct Camera {
private:
    Vec3 position;
    float aspectRatio;
    int pixelWidth, pixelHeight, halfPixelWidth, halfPixelHeight;
    // Perpendicular distance from the camera position to the projection

```

```

plane
    float distToProjPlane;

    int lastModelIndex = 0;
    // Internal list of models
    std::vector<std::shared_ptr<Model>> models;

    void renderPixel(SDL_Renderer* renderer, int pixelX, int pixelY);
    Ray emitScreenRay(int pixelX, int pixelY);
    // Get the colour of whatever a given ray intersects with
    Vec3 getRayIntersectionColour(Ray& ray);
    // Get the indices of the model and triangle that a given ray
intersects with
    void getCollisionIndices(Ray& ray, int& modelIndex, int&
triangleIndex);
    // Get the brightness of a surface at a given point
    float getBrightnessOfSurface(int& modelIndex, int& triangleIndex);
    // Set the drawing colour of the pixels in the SDL window
    void setDrawColour(SDL_Renderer* renderer, Vec3& colour);
    // Tone-map HDR (high dynamic range) colours to LDR (low dynamic
range)
    // colours that can be represented on a screen
    float toneMap(float value);
public:
    // Default constructor
    Camera(Vec3 _position, int _pixelWidth, int _pixelHeight, float
_horizontalFOV = 90.0f);

    void renderImage(SDL_Renderer* renderer, int screenWidth, int
screenHeight);
    // Add a model to the camera's internal list
    void insertModel(std::shared_ptr<Model> object);
};

```

camera.cpp

```

#include <iostream>
#include <chrono>
#include <algorithm>

```



```

#include "camera.h"

// Various constants
static constexpr float PI = 3.14159265;
static constexpr float DEG2RAD = PI / 180;
static constexpr float MAX_DIST = 1000000.0;

// Default constructor
Camera::Camera(Vec3 _position, int _pixelWidth, int _pixelHeight, float
_horizontalFOV)
    : position(_position), pixelWidth(_pixelWidth),
pixelHeight(_pixelHeight) {
    aspectRatio = pixelWidth / (float)pixelHeight;
    float verticalFOV = _horizontalFOV / aspectRatio;
    float halfFOV = _horizontalFOV / 2.0f;
    // The distance to the projection plane is the cotangent of half the
horizontal field of view
    distToProjPlane = 1 / tan(halfFOV * DEG2RAD); // Converting degrees
to radians
    halfPixelWidth = pixelWidth / 2;
    halfPixelHeight = pixelHeight / 2;
}

void Camera::renderImage(SDL_Renderer* renderer, int screenWidth, int
screenHeight) {
    time_t start = time(0);
    // Iterate over each pixel in the screen, emitting a ray for each
    for (int y = 0; y < screenHeight; y++) {
        for (int x = 0; x < screenWidth; x++) {
            renderPixel(renderer, x, y);
        }
        // Update the screen with the new row
        SDL_RenderPresent(renderer);
    }
}

void Camera::renderPixel(SDL_Renderer* renderer, int pixelX, int pixelY) {
    // Emit a ray into the scene, and get the colour of whatever it
collides with
    Ray ray = emitScreenRay(pixelX, pixelY);
    Vec3 colour = getRayIntersectionColour(ray);
    // Set the draw colour, and draw it to the required pixel

```

```

        setDrawColour(renderer, colour);
        SDL_RenderDrawPoint(renderer, pixelX, pixelY);
    }

Ray Camera::emitScreenRay(int pixelX, int pixelY) {
    // Get the coordinates of the ray in view-space coordinates
    float screenX = (pixelX - halfPixelWidth) / (float)halfPixelWidth;
    float screenY = (pixelY - halfPixelHeight) / (float)halfPixelHeight;
    // The coordinates are distributed in a square, so scale the
    x-coordinate
    // by the aspect ratio to get the correct proportions in world-space
    screenX *= aspectRatio;
    // Find the normalised ray direction in world space
    Vec3 rayDirection = Vec3(screenX, screenY,
distToProjPlane).normalise();
    return Ray(position, rayDirection);
}

// Get the colour of whatever a given ray intersects with
Vec3 Camera::getRayIntersectionColour(Ray& ray) {
    Vec3 intersection = Vec3();
    // Indices are initialised to '-1' to detect if no collision occurs
    int modelIndex = -1;
    int triangleIndex = -1;
    // Get the index of the triangle (and it's parent model)
    // that the ray has it's closest intersection with
    getCollisionIndices(ray, modelIndex, triangleIndex);
    // Initially set the pixel colour to the background colour
    Vec3 colour = Vec3(0.02, 0.02, 0.04);

    if (modelIndex != -1 && triangleIndex != -1) {
        // Find the brightness of the point on the triangle
        float brightness = getBrightnessOfSurface(modelIndex,
triangleIndex);
        colour = models[modelIndex]->colour * brightness;
    }
    return colour;
}

// Get the indices of the model and triangle that a given ray intersects
with
void Camera::getCollisionIndices(Ray& ray, int& modelIndex, int&

```

```

triangleIndex) {
    float t = (float)MAX_DIST;
    float closest = t * 2;
    int tempTriangleIndex = -1;
    // Iterate over models, checking for collisions
    // If there is a successful intersection and it's the closest one
    yet, set all the out variables
    for (int i = 0; i < lastModelIndex; i++) {
        bool isIntersection = models[i]->rayIntersection(ray, t,
tempTriangleIndex);
        if (isIntersection && t < closest) {
            modelIndex = i;
            triangleIndex = tempTriangleIndex;
            closest = t;
        }
    }
}

// Get the brightness of a surface at a given point
float Camera::getBrightnessOfSurface(int& modelIndex, int& triangleIndex) {
    // Get the normal vector to the triangle, and use it to calculate the
    brightness at that point
    Triangle triangle = models[modelIndex]->getTriangle(triangleIndex);
    int normalIndex = triangle.getNormalIndex();
    Ray normalRay = Ray(
        models[modelIndex]->getPosition(),
        models[modelIndex]->getNormal(normalIndex));
    // Get brightness by angle towards positive x-axis
    // Brightness is in the range [0, 1] so raising
    // to a power of 3 creates sharper highlights
    float brightness = (normalRay.getDirection().dot(Vec3(1.0, 0.0, 0.0))
/ 2.0) + 0.5;
    return pow(brightness, 3.0);
}

// Set the drawing colour of the pixels in the SDL window
void Camera::setDrawColour(SDL_Renderer* renderer, Vec3& colour) {
    // Tone-map the colour, and convert each component to an 8 bit
    integer
    SDL_SetRenderDrawColor(
        renderer,
        (Uint8)(toneMap(colour.x) * 256),

```

```

        (UInt8)(toneMap(colour.y) * 256),
        (UInt8)(toneMap(colour.z) * 256),
        (UInt8)255);
    }

    // Tone-map HDR (high dynamic range) colours to LDR (low dynamic range)
    // colours that can be represented on a screen
    float Camera::toneMap(float value) {
        float mapped = value / (value + 1);
        return pow(mapped, 1 / 2.2); // Gamma value
    }

    // Add a model to the camera's internal list
    void Camera::insertModel(std::shared_ptr<Model> model) {
        models.push_back(model);
        lastModelIndex += 1;
    }

```

model.h

```

#pragma once

struct BVHNode;

#include <vector>
#include <memory>
#include "BVH.h"
#include "modelloader.h"

struct Model {
private:
    Vec3 position;
    std::vector<Triangle> triangles;
    std::vector<Vec3> vertices;
    std::vector<Vec3> normals;
    std::shared_ptr<BVHNode> rootNode;
public:
    Vec3 colour = Vec3(1.0f, 0.0f, 0.0f);

```

```

// Default constructor
Model(
    std::string filePath,
    Vec3 _position = Vec3(),
    Transform transform = Transform());
// Copy constructor
Model(const Model& _object);
Triangle getTriangle(int index) const;
Vec3 getPosition() const;
Vec3 getVertex(int index) const;
int getVertexNum() const;
Vec3 getNormal(int index) const;
// Get the triangle, and distance to intersection of a given ray with
the model
bool rayIntersection(const Ray& ray, float& t, int& triangleIndex)
const;
};

struct Triangle {
private:
    uint32_t v0Index, v1Index, v2Index;
    uint32_t normalIndex;
    uint32_t triangleIndex;
    std::shared_ptr<Model> parent;
    float planeOffset;
    Vec3 center;
public:
    // Default constructor
    Triangle(
        uint32_t _v0Index = -1,
        uint32_t _v1Index = -1,
        uint32_t _v2Index = -1,
        uint32_t _normalIndex = -1,
        uint32_t _triangleIndex = -1,
        std::shared_ptr<Model> _parent = nullptr);
    // Copy constructor
    Triangle(const Triangle& other);
    // Calculate distance to intersection point of ray and triangle
    bool rayIntersection(const Ray& ray, const Vec3& offset, float& t)
const;
    void setParent(std::shared_ptr<Model> parent);
    int getv0Index() const;

```

```

    int getv1Index() const;
    int getv2Index() const;
    int getNormalIndex() const;
    int getTriangleIndex() const;
    Vec3 getCenter() const;
};

```

model.cpp

```

#include "model.h"

// ----- //
//           Model           //
// ----- //

// Default constructor
Model::Model(
    std::string filePath,
    Vec3 _position,
    Transform transform)
    : position(_position) {
    // Temporary lists for vertex and normal indices
    std::vector<uint32_t> vertexIndices;
    std::vector<uint32_t> normalIndices;
    // Load model data from an OBJ file into the various lists
    loadOBJ(filePath.c_str(), vertices, normals, vertexIndices,
normalIndices, transform);
    int triNum = normalIndices.size() / 3;
    for (int i = 0; i < triNum; i++) {
        // Triangles have 3 vertices, and vertices are stored
        // in a contiguous list, so use a step size of 3
        uint32_t index = i * 3;
        // Triangles store only the indices of their
        // attributes, not the attributes themselves
        triangles.push_back(
            Triangle(
                vertexIndices[index] - 1,
                vertexIndices[index + 1] - 1,
                vertexIndices[index + 2] - 1,
                normalIndices[index] - 1,

```

```

        i,
        std::shared_ptr<Model>(this)
    ));
}

// Create the root node of the BVH of the model, which then creates
it's own hierarchy
    rootNode = std::make_shared<BVHNode>(triangles, this);
}

// Copy constructor
Model::Model(const Model& _model)
    : position(_model.position),
      triangles(_model.triangles),
      vertices(_model.vertices),
      normals(_model.normals),
      rootNode(std::move(_model.rootNode)) {

Triangle Model::getTriangle(int index) const {
    return triangles[index];
}

Vec3 Model::getVertex(int index) const {
    return vertices[index];
}

int Model::getVertexNum() const {
    return vertices.size();
}

Vec3 Model::getNormal(int index) const {
    return normals[index];
}

// Get the triangle, and distance to intersection of a given ray with the
model
bool Model::rayIntersection(const Ray& ray, float& t, int& triangleIndex)
const {
    return rootNode->rayIntersection(ray, t, triangleIndex);
}

Vec3 Model::getPosition() const {

```

```

        return position;
    }

// ----- //
//             Triangle             //
// ----- //

// Default constructor
Triangle::Triangle(
    uint32_t _v0Index, uint32_t _v1Index, uint32_t _v2Index,
    uint32_t _normalIndex, uint32_t _triangleIndex,
    std::shared_ptr<Model> _parent)
    : v0Index(_v0Index), v1Index(_v1Index), v2Index(_v2Index),
      normalIndex(_normalIndex), triangleIndex(_triangleIndex),
      parent(_parent) {

    planeOffset =
        -parent->getNormal(normalIndex).dot(parent->getVertex(v0Index));
    center = (parent->getVertex(v0Index) + parent->getVertex(v1Index) +
        parent->getVertex(v2Index)) / 3.0f;
}

// Copy constructor
Triangle::Triangle(const Triangle& other)
    : v0Index(other.v0Index), v1Index(other.v1Index),
      v2Index(other.v2Index),
      normalIndex(other.normalIndex), triangleIndex(other.triangleIndex),
      parent(other.parent), planeOffset(other.planeOffset),
      center(other.center) {
}

// Calculate distance to intersection point of ray and triangle
bool Triangle::rayIntersection(const Ray& ray, const Vec3& offset, float&
t) const {
    // Möller-Trumbore algorithm
    //
https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection
    //
http://www.lighthouse3d.com/tutorials/maths/ray-triangle-intersection/
    Vec3 rayOrigin = ray.getOrigin();
    Vec3 rayDirection = ray.getDirection();

```



```

// Offset is the position of the model in world space,
// used to transform the vertex coordinates to world
// space, as they are stored as model space coordinates
Vec3 v0 = parent->getVertex(v0Index) + offset;
Vec3 v1 = parent->getVertex(v1Index) + offset;
Vec3 v2 = parent->getVertex(v2Index) + offset;

Vec3 edge0 = v1 - v0;
Vec3 edge1 = v2 - v0;
Vec3 pvec = rayDirection.cross(edge1);
float det = edge0.dot(pvec);
// If the determinant is negative the triangle is backfacing
if (det <= 0.00001f) return false;
float invDet = 1 / det;

Vec3 tvec = rayOrigin - v0;
// 'u' component of the barycentric coordinates
float u = tvec.dot(pvec) * invDet;
// If 'u' is outside the range [0,1] then
// the intersection is outside the triangle
if (u < 0.0f || u > 1.0f) return false;
Vec3 qvec = tvec.cross(edge0);
// 'v' component of the barycentric coordinates
float v = rayDirection.dot(qvec) * invDet;
// If 'v' is negative or 'u+v' is more than 1 then
// the intersection is outside the triangle
if (v < 0.0f || u + v > 1.0f) return false;
t = edge1.dot(qvec) * invDet;
return true;
}

void Triangle::setParent(std::shared_ptr<Model> _parent) {
    parent = _parent;
}

int Triangle::getv0Index() const {
    return v0Index;
}

int Triangle::getv1Index() const {
    return v1Index;
}

```

```

int Triangle::getv2Index() const {
    return v2Index;
}

int Triangle::getNormalIndex() const {
    return normalIndex;
}

int Triangle::getTriangleIndex() const {
    return triangleIndex;
}

Vec3 Triangle::getCenter() const {
    return center;
}

```

modelloader.h

```

#pragma once

#include <vector>
#include "transform.h"

// Load vertices, normals, and faces from an OBJ file at 'path'
bool loadOBJ(
    const char* path,
    std::vector<Vec3>& outVertices,
    std::vector<Vec3>& outNormals,
    std::vector<uint32_t>& outVertexIndices,
    std::vector<uint32_t>& outNormalIndices,
    const Transform& transform);

// Parse vertex data from a line in the file
static void parseVertex(
    FILE* file,
    std::vector<Vec3>& tempVertices,
    const Transform& transform);

// Parse normal data from a line in the file
static void parseNormal(

```

```

        FILE* file,
        std::vector<Vec3>& tempNormals,
        const Transform& transform);

// Parse a face from a line in the file
// A face is given by indices to vertices and normals
static void parseFace(FILE* file,
        std::vector<uint32_t>& vertexIndices,
        std::vector<uint32_t>& normalIndices,
        const Transform& transform);

// Parse vertices, normals, and faces from an OBJ file
static void parseOBJ(FILE* file,
        std::vector<uint32_t>& vertexIndices,
        std::vector<uint32_t>& normalIndices,
        std::vector<Vec3>& tempVertices,
        std::vector<Vec3>& tempNormals,
        const Transform& transform);

```

modelloader.cpp

```

// Disable warnings for unsafe file I/O
#define _CRT_SECURE_NO_WARNINGS

#include <Python.h>
#include <Windows.h>
#include <string>

#include "modelloader.h"

// Output to Python console
static void print(std::string str) {
    PySys_WriteStdout(str.c_str());
    PySys_WriteStdout("\n");
}

// Load vertices, normals, and faces from an OBJ file at 'path'
bool loadOBJ(
    const char* path,

```

```

        std::vector<Vec3>& outVertices,
        std::vector<Vec3>& outNormals,
        std::vector<uint32_t>& outVertexIndices,
        std::vector<uint32_t>& outNormalIndices,
        const Transform& transform
    ) {
        // Get file handle
        FILE* file = fopen(path, "r");
        // Validate file handle
        if (file == nullptr) {
            print("Impossible to open OBJ file!");
            print(std::string(path));
            return false;
        }
        // Parse from file
        parseOBJ(file,
            outVertexIndices, outNormalIndices,
            outVertices, outNormals, transform);
        return true;
    }

    // Parse vertex data from a line in the file
    static void parseVertex(
        FILE* file,
        std::vector<Vec3>& tempVertices,
        const Transform& transform
    ) {
        Vec3 vertex;
        fscanf_s(file, "%f %f %f\n", &vertex.x, &vertex.y, &vertex.z);
        tempVertices.push_back(transform.transform(vertex));
    }

    // Parse normal data from a line in the file
    static void parseNormal(
        FILE* file,
        std::vector<Vec3>& tempNormals,
        const Transform& transform
    ) {
        Vec3 normal;
        fscanf_s(file, "%f %f %f\n", &normal.x, &normal.y, &normal.z);
        tempNormals.push_back(transform.transform(normal));
    }

```

```

// Parse a face from a line in the file
// A face is given by indices to vertices and normals
static void parseFace(
    FILE* file,
    std::vector<unsigned int>& vertexIndices,
    std::vector<unsigned int>& normalIndices,
    const Transform& transform
) {
    std::string vertex1, vertex2, vertex3;
    unsigned int vertexIndex[3], normalIndex[3];
    int matches = fscanf_s(
        file, "%d//%d %d//%d %d//%d\n",
        &vertexIndex[0], &normalIndex[0],
        &vertexIndex[1], &normalIndex[1],
        &vertexIndex[2], &normalIndex[2]);
    if (matches != 6) {
        print("File can't be read, try exporting with other
options\n");
        return;
    }
    // OBJ files are exported with the wrong
    // winding order, this flips them
    int v0IndexIndex = 0;
    int v1IndexIndex = 1;
    int v2IndexIndex = 2;
    transform.flipVertexIndices(
        v1IndexIndex,
        v2IndexIndex);
    // Add indices to their respective lists
    vertexIndices.push_back(vertexIndex[v0IndexIndex]);
    vertexIndices.push_back(vertexIndex[v1IndexIndex]);
    vertexIndices.push_back(vertexIndex[v2IndexIndex]);
    normalIndices.push_back(normalIndex[0]);
    normalIndices.push_back(normalIndex[1]);
    normalIndices.push_back(normalIndex[2]);
}

// Parse vertices, normals, and faces from an OBJ file
static void parseOBJ(
    FILE* file,
    std::vector<unsigned int>& vertexIndices,

```

```

        std::vector<unsigned int>& normalIndices,
        std::vector<Vec3>& tempVertices,
        std::vector<Vec3>& tempNormals,
        const Transform& transform
    ) {
        // Iterate through the lines in the file
        while (true) {
            char lineHeader[128];
            int res = fscanf(file, "%s", lineHeader);
            // If the end of the file is reached, break the loop
            if (res == EOF) break;
            // If a line begins with a 'v', it designates a vertex
            if (strcmp(lineHeader, "v") == 0) {
                parseVertex(file, tempVertices, transform);
            }
            // If a line begins with a 'vn', it designates a normal
            else if (strcmp(lineHeader, "vn") == 0) {
                parseNormal(file, tempNormals, transform);
            }
            // If a line begins with an 'f', it designates a face
            else if (strcmp(lineHeader, "f") == 0) {
                parseFace(file, vertexIndices, normalIndices, transform);
            }
        }
    }
}

```

BVH.h

```

#pragma once

struct Model;
struct Triangle;

#include "geometry.h"
#include "model.h"

// Used to identify specific axes
namespace Axes {
    enum Axes {
        x, y, z
    };
}

```

```

    };
}

struct BVHNode {
private:
    static constexpr int maxTriangleNumPerLeaf = 3;

    std::unique_ptr<BVHNode> child0;
    std::unique_ptr<BVHNode> child1;
    std::vector<Triangle> triangles;
    Vec3 center = Vec3();
    Vec3 modelOffset;
    float radius;
    bool isLeaf;

    // Update the radius of the spherical bounds of the
    // node so that the given vertex is within the bounds
    void updateBoundRadius(Vec3 vertex);
    // Calculate the minimum radius of the spherical bounds of a model
    void calcBounds(Model* Model);
    // Calculate the variance of the positions of the triangle centers
    // in each axis, and return the axis with the greatest variance
    Axes::Axes calcAxisWithGreatestVariance();
    // Partition the triangles into two groups along the x axis
    void partitionX(
        std::vector<Triangle>& leftPartitionTriangles,
        std::vector<Triangle>& rightPartitionTriangles);
    // Partition the triangles into two groups along the y axis
    void partitionY(
        std::vector<Triangle>& leftPartitionTriangles,
        std::vector<Triangle>& rightPartitionTriangles);
    // Partition the triangles into two groups along the z axis
    void partitionZ(
        std::vector<Triangle>& leftPartitionTriangles,
        std::vector<Triangle>& rightPartitionTriangles);
    // Partition the triangles into two groups
    void partition(Model* Model);
    // Calculate the distance to the intersection
    // point between a ray and the bounding sphere
    bool raySphereIntersection(const Ray& ray);
    // Find the closest intersection between a ray and the triangles
    bool rayTrianglesIntersection(const Ray& ray, float& t, int&

```

```

triangleIndex);
    // Recursively calculate ray intersections between the heirarchy of
    bounding volumes
    bool recurseRayIntersection(const Ray& ray, float& t, int&
triangleIndex);
public:
    // Default constructor
    BVHNode(std::vector<Triangle> triangles, Model* Model);
    // If the number of triangles is above the maximum, partition,
    // else identify itself as a leaf node in the tree
    void build(Model* Model);
    // If the ray intersects with the bounding volume, intersect with the
    children or triangles
    bool rayIntersection(const Ray& ray, float& t, int& triangleIndex);
};

```

BVH.cpp

```

#include "BVH.h"

static constexpr float MAX_DIST = 1000000.0;

// Update the radius of the spherical bounds of the
// node so that the given vertex is within the bounds
void BVHNode::updateBoundRadius(Vec3 vertex) {
    const float tempRadius = (vertex - center).getLength();
    if (radius < tempRadius) {
        radius = tempRadius;
    }
}

// Calculate the minimum radius of the spherical bounds of a model
void BVHNode::calcBounds(Model* model) {
    // Calculate the average center of all the triangles in the set
    for (int i = 0; i < triangles.size(); i++) {
        center = center + triangles[i].getCenter();
    }
    center = center / triangles.size();
    // Update the bounds, testing each vertex of every triangle in the
    set
    for (int i = 0; i < triangles.size(); i++) {

```



```

        updateBoundRadius(
            model->getVertex(
                triangles[i].getv0Index()));
        updateBoundRadius(
            model->getVertex(
                triangles[i].getv1Index()));
        updateBoundRadius(
            model->getVertex(
                triangles[i].getv2Index()));
    }
}

// Get the axis with the greatest variance
Axes::Axes getAxis(Vec3 variance) {
    // Variance is stored in a 3-vector ordered by axis
    if (variance.x > variance.y && variance.x > variance.z) return
Axes::x;
    else if (variance.y > variance.z) return
Axes::y;
    else return
Axes::z;
}

// Calculate the variance of the positions of the triangle centers
// in each axis, and return the axis with the greatest variance
Axes::Axes BVHNode::calcAxisWithGreatestVariance() {
    Vec3 mean, sumOfSqrs;
    for (int i = 0; i < triangles.size(); i++) {
        // Using Welford's method for computing variance
        Vec3 center = triangles[i].getCenter();
        Vec3 oldMean = mean;
        mean = mean + (center - mean) / (float)(i + 1);
        sumOfSqrs = sumOfSqrs + (center - mean) * (center - oldMean);
    }
    return getAxis(sumOfSqrs);
}

// Partition the triangles into two groups along the x axis
void BVHNode::partitionX(std::vector<Triangle>& leftPartitionTriangles,
std::vector<Triangle>& rightPartitionTriangles) {
    for (int i = 0; i < triangles.size(); i++) {
        if (triangles[i].getCenter().x < center.x) {

```

```

        leftPartitionTriangles.push_back(triangles[i]);
    }
    else {
        rightPartitionTriangles.push_back(triangles[i]);
    }
}

// Partition the triangles into two groups along the y axis
void BVHNode::partitionY(std::vector<Triangle>& leftPartitionTriangles,
std::vector<Triangle>& rightPartitionTriangles) {
    for (int i = 0; i < triangles.size(); i++) {
        if (triangles[i].getCenter().y < center.y) {
            leftPartitionTriangles.push_back(triangles[i]);
        }
        else {
            rightPartitionTriangles.push_back(triangles[i]);
        }
    }
}

// Partition the triangles into two groups along the z axis
void BVHNode::partitionZ(std::vector<Triangle>& leftPartitionTriangles,
std::vector<Triangle>& rightPartitionTriangles) {
    for (int i = 0; i < triangles.size(); i++) {
        if (triangles[i].getCenter().z < center.z) {
            leftPartitionTriangles.push_back(triangles[i]);
        }
        else {
            rightPartitionTriangles.push_back(triangles[i]);
        }
    }
}

// Partition the triangles into two groups
void BVHNode::partition(Model* model) {
    std::vector<Triangle> leftPartitionTriangles;
    std::vector<Triangle> rightPartitionTriangles;
    Axes::Axes greatestVarianceAxis = calcAxisWithGreatestVariance();
    // Partition along the axis with the greatest variance in triangle
    position
    switch (greatestVarianceAxis) {

```

```

    case Axes::x:
        partitionX(leftPartitionTriangles, rightPartitionTriangles);
        break;
    case Axes::y:
        partitionY(leftPartitionTriangles, rightPartitionTriangles);
        break;
    case Axes::z:
        partitionZ(leftPartitionTriangles, rightPartitionTriangles);
        break;
}
// Initialise the children of the node using the two separated groups
child0 = std::make_unique<BVHNode>(leftPartitionTriangles, model);
child1 = std::make_unique<BVHNode>(rightPartitionTriangles, model);
}

// Calculate the distance to the intersection
// point between a ray and the bounding sphere
bool BVHNode::raySphereIntersection(const Ray& ray) {
    float t0, t1;
    const float radius2 = radius * radius;
    const Vec3 l = center + modelOffset - ray.getOrigin();
    // Parallel distance to sphere center
    const float tca = l.dot(ray.getDirection());
    // Perpendicular distance to sphere center
    const float d2 = l.dot(l) - tca * tca;
    // Ray doesn't intersect with sphere
    if (d2 > radius2) return false;
    // Distance from center to intersection parallel to ray
    const float thc = sqrt(radius2 - d2);
    // Calculate distance to first and second intersection point
    t0 = tca - thc;
    t1 = tca + thc;
    // If t0 is in front of t1, swap the two
    if (t0 > t1) std::swap(t0, t1);
    if (t1 < 0) {
        // Both intersections are behind the ray's origin
        return false;
    }
    return true;
}

// Find the closest intersection between a ray and the triangles

```

```

bool BVHNode::rayTrianglesIntersection(const Ray& ray, float& t, int&
triangleIndex) {
    bool isIntersection = false;
    for (int i = 0; i < triangles.size(); i++) {
        float dist = MAX_DIST;
        if (triangles[i].rayIntersection(ray, modelOffset, dist)) {
            if (dist < t) {
                t = dist;
                triangleIndex = triangles[i].getTriangleIndex();
                isIntersection = true;
            }
        }
    }
    return isIntersection;
}

```

// Recursively calculate ray intersections between the heirarchy of bounding volumes

```

bool BVHNode::recurseRayIntersection(const Ray& ray, float& t, int&
triangleIndex) {
    int triangleIndex0 = -1;
    int triangleIndex1 = -1;
    float dist0 = MAX_DIST * 2;
    float dist1 = MAX_DIST * 2;
    bool isIntersect0 = child0->rayIntersection(ray, dist0,
triangleIndex0);
    bool isIntersect1 = child1->rayIntersection(ray, dist1,
triangleIndex1);
    if (dist0 > dist1) {
        std::swap(dist0, dist1);
        std::swap(triangleIndex0, triangleIndex1);
    }
    if (dist0 < t) {
        t = dist0;
        triangleIndex = triangleIndex0;
    }
    return isIntersect0 || isIntersect1;
}

```

// Default constructor

```

BVHNode::BVHNode(const std::vector<Triangle> _triangles, Model* model)
: triangles(_triangles) {

```

```

        calcBounds(model);
        build(model);
        modelOffset = model->getPosition();
    }

    // If the number of triangles is above the maximum, partition,
    // else identify itself as a leaf node in the tree
    void BVHNode::build(Model* model) {
        isLeaf = true;
        if (triangles.size() > maxTriangleNumPerLeaf) {
            partition(model);
            // Free memory
            std::vector<Triangle>().swap(triangles);
            triangles.clear();
            isLeaf = false;
        }
    }

    // If the ray intersects with the bounding volume, intersect with the
    // children or triangles
    bool BVHNode::rayIntersection(const Ray& ray, float& t, int& triangleIndex)
    {
        if (!raySphereIntersection(ray)) {
            return false;
        }
        else if (isLeaf) {
            return rayTrianglesIntersection(ray, t, triangleIndex);
        }
        else {
            return recurseRayIntersection(ray, t, triangleIndex);
        }
    }

```

transform.h

```

#pragma once

#include "geometry.h"
#include <math.h>
#include <algorithm>

```

```

struct Transform {
    // Default constructor
    Transform(
        float rotX = 0.0f,
        float rotY = 0.0f,
        float rotZ = 0.0f,
        bool _flipX = false,
        bool _flipY = false,
        bool _flipZ = false
    );
    // Copy constructor
    Transform(const Transform& other);
    // Take a vector, and apply the transformation matrix to it
    Vec3 transform(const Vec3& vector) const;
    // Swap two vertices in a triangle to flip the winding order
    void flipVertexIndices(int& v0Index, int& v1Index) const;
private:
    Mat3 mat;
    bool flipX, flipY, flipZ;

    // Calculate the required rotation matrix to rotate by the given
    angles in each axis
    Mat3 calcRotMat(const float rotX, const float rotY, const float rotZ)
const;
    // Calculate the required reflection matrix, to mirror in the given
    axes
    Mat3 calcRefMat(const bool flipX, const bool flipY, const bool flipZ)
const;
};

```

transform.cpp

```

#include "transform.h"

// Various constants
constexpr float PI = 3.14159265;
constexpr float DEG2RAD = PI / 180.0;

// Default constructor
Transform::Transform(float rotX, float rotY, float rotZ, bool _flipX, bool

```

```

_flipY, bool _flipZ)
    : flipX(_flipX), flipY(_flipY), flipZ(_flipZ) {
    Mat3 rotMat = calcRotMat(rotX, rotY, rotZ);
    Mat3 refMat = calcRefMat(_flipX, _flipY, _flipZ);
    // Combine the rotation and reflection matrices to create a single
    transformation
    // Order of operands matters here
    mat = rotMat * refMat;
}

// Copy constructor
Transform::Transform(const Transform& other)
    : mat(other.mat) {
}

// Take a vector, and apply the transformation matrix to it
Vec3 Transform::transform(const Vec3& vec) const {
    return mat * vec;
}

// Swap two vertices in a triangle to flip the winding order
void Transform::flipVertexIndices(int& v0Index, int& v1Index) const {
    bool flipOneAxis = flipX != flipY != flipZ;
    bool flipAllAxes = flipX && flipY && flipZ;
    if (flipOneAxis || flipAllAxes) { // XOR or all equal
        std::swap(v0Index, v1Index);
    }
}

// Calculate the required rotation matrix to rotate by the given angles in
each axis
Mat3 Transform::calcRotMat(const float rotX, const float rotY, const float
rotZ) const {
    // Cache trigonometric results, as they're expensive to calculate
    // and are used multiple times (literally doesn't even matter)
    float cx = cos(rotX * DEG2RAD);
    float sx = sin(rotX * DEG2RAD);
    float cy = cos(rotY * DEG2RAD);
    float sy = sin(rotY * DEG2RAD);
    float cz = cos(rotZ * DEG2RAD);
    float sz = sin(rotZ * DEG2RAD);
    // Calculate individual rotation matrices for each axis

```

```

    Mat3 rotXMat = Mat3(
        1.0f, 0.0f, 0.0f,
        0.0f, cx, -sx,
        0.0f, sx, cx);
    Mat3 rotYMat = Mat3(
        cy, 0.0f, sy,
        0.0f, 1.0f, 0.0f,
        -sy, 0.0f, cy);
    Mat3 rotZMat = Mat3(
        cz, -sz, 0.0f,
        sz, cz, 0.0f,
        0.0f, 0.0f, 1.0f);
    // Combine all three matrices into one transformation
    // Order of operands matters here
    return rotXMat * rotYMat * rotZMat;
}

// Calculate the required reflection matrix, to mirror in the given axes
Mat3 Transform::calcRefMat(const bool flipX, const bool flipY, const bool
flipZ) const {
    // Just a little formula for mapping 0 to 1 and 1 to -1
    // Taking advantage of the fact that booleans implicitly cast to
    integers
    return Mat3(
        1 - 2 * flipX, 0, 0,
        0, 1 - 2 * flipY, 0,
        0, 0, 1 - 2 * flipZ
    );
}

```

geometry.h

```

#pragma once

struct Vec3 {
    float x, y, z;

    // Default constructor
    Vec3(float _x = 0.0f, float _y = 0.0f, float _z = 0.0f);
    // Copy constructor
    Vec3(const Vec3& other);
}

```



```

// Dot product
float dot(const Vec3& other) const;
// Cross product
Vec3 cross(const Vec3& other) const;
// Addition operator
Vec3 operator+(const Vec3& other) const;
// Subtraction operator
Vec3 operator-(const Vec3& other) const;
// Vector-scalar multiplication operator
Vec3 operator*(const float other) const;
// Component-wise vector-vector multiplication operator
Vec3 operator*(const Vec3& other) const;
// Vector-scalar division operator
Vec3 operator/(const float other) const;
// Get the magnitude of the vector
float getLength() const;
// Return a normalised version of the vector (i.e of magnitude 1)
Vec3 normalise() const;
// Get the vector in the opposite direction with the same magnitude
Vec3 getReverse() const;
};

struct Mat3 {
    float x0, y0, z0, x1, y1, z1, x2, y2, z2;

    // Default constructor
    Mat3(
        float _x0 = 1.0f, float _y0 = 0.0f, float _z0 = 0.0f,
        float _x1 = 0.0f, float _y1 = 1.0f, float _z1 = 0.0f,
        float _x2 = 0.0f, float _y2 = 0.0f, float _z2 = 1.0f);
    // Copy constructor
    Mat3(const Mat3& other);
    // Get the matrix multiplied by itself
    Mat3 getSquare() const;
    // Addition operator
    Mat3 operator+(const Mat3& other) const;
    // Matrix-scalar multiplication operator
    Mat3 operator*(const float other) const;
    // Matrix-vector multiplication operator
    Vec3 operator*(const Vec3& other) const;
    // Matrix-matrix multiplication operator
    Mat3 operator*(const Mat3& other) const;

```

```

};

struct Ray {
private:
    Vec3 origin;
    Vec3 direction;
public:
    // Default constructor
    Ray(Vec3 _origin = Vec3(), Vec3 _direction = Vec3(1.0f, 0.0f, 0.0f));
    // Copy constructor
    Ray(const Ray& other);
    // Get the origin of the ray
    Vec3 getOrigin() const;
    // Get the direction of the ray
    Vec3 getDirection() const;
    // Set the origin of the ray
    void setOrigin(const Vec3& _origin);
    // Set the direction of the ray
    void setDirection(const Vec3& _direction);
    // Normalise the direction of the ray
    void normalise();
    // Get the point at distance 't' from the origin along the direction
    vector
    Vec3 project(const float t) const;
};

```

geometry.cpp

```

#include <algorithm>
#include "geometry.h"

// ----- //
//           Vector 3           //
// ----- //

// Default constructor
Vec3::Vec3(float _x, float _y, float _z)
    : x(_x), y(_y), z(_z) {
}
// Copy constructor

```

```

Vec3::Vec3(const Vec3& other)
    : x(other.x), y(other.y), z(other.z) {
}
// Dot product
float Vec3::dot(const Vec3& other) const {
    return x * other.x + y * other.y + z * other.z;
}
// Cross product
Vec3 Vec3::cross(const Vec3& other) const {
    return Vec3(
        y * other.z - z * other.y,
        z * other.x - x * other.z,
        x * other.y - y * other.x);
}
// Addition operator
Vec3 Vec3::operator+(const Vec3& other) const {
    return Vec3(x + other.x, y + other.y, z + other.z);
}
// Subtraction operator
Vec3 Vec3::operator-(const Vec3& other) const {
    return Vec3(x - other.x, y - other.y, z - other.z);
}
// Vector-scalar multiplication operator
Vec3 Vec3::operator*(const float other) const {
    return Vec3(x * other, y * other, z * other);
}
// Component-wise vector-vector multiplication operator
Vec3 Vec3::operator*(const Vec3& other) const {
    return Vec3(x * other.x, y * other.y, z * other.z);
}
// Vector-scalar division operator
Vec3 Vec3::operator/(const float other) const {
    return Vec3(x / other, y / other, z / other);
}
// Get the magnitude of the vector
float Vec3::getLength() const {
    return sqrt(x * x + y * y + z * z);
}
// Return a normalised version of the vector (i.e of magnitude 1)
Vec3 Vec3::normalise() const {
    float length = getLength();
    return Vec3(x / length, y / length, z / length);
}

```

```

}
// Get the vector in the opposite direction with the same magnitude
Vec3 Vec3::getReverse() const {
    return Vec3(-x, -y, -z);
}

// ----- //
//           Matrix 3x3           //
// ----- //

// Default constructor
Mat3::Mat3(float _x0, float _y0, float _z0, float _x1, float _y1, float
_z1, float _x2, float _y2, float _z2)
    : x0(_x0), y0(_y0), z0(_z0), x1(_x1), y1(_y1), z1(_z1), x2(_x2),
y2(_y2), z2(_z2) {
}
// Copy constructor
Mat3::Mat3(const Mat3& other)
    : x0(other.x0), y0(other.y0), z0(other.z0),
x1(other.x1), y1(other.y1), z1(other.z1),
x2(other.x2), y2(other.y2), z2(other.z2) {
}
// Get the matrix multiplied by itself
Mat3 Mat3::getSquare() const {
    return Mat3(
        x0 * x0 + y0 * x1 + z0 * x2,
        x0 * y0 + y0 * y1 + z0 * y2,
        x0 * z0 + y0 * z1 + z0 * z2,
        x1 * x0 + y1 * x1 + z1 * x2,
        x1 * y0 + y1 * y1 + z1 * y2,
        x1 * z0 + y1 * z1 + z1 * z2,
        x2 * x0 + y2 * x1 + z2 * x2,
        x2 * y0 + y2 * y1 + z2 * y2,
        x2 * z0 + y2 * z1 + z2 * z2);
}
// Addition operator
Mat3 Mat3::operator+(const Mat3& other) const {
    return Mat3(
        x0 + other.x0, y0 + other.y0, z0 + other.z0,
        x1 + other.x1, y1 + other.y1, z1 + other.z1,
        x2 + other.x2, y2 + other.y2, z2 + other.z2);
}

```

```

// Matrix-scalar multiplication operator
Mat3 Mat3::operator*(const float other) const {
    return Mat3(
        x0 * other, y0 * other, z0 * other,
        x1 * other, y1 * other, z1 * other,
        x2 * other, y2 * other, z2 * other);
}

// Matrix-vector multiplication operator
Vec3 Mat3::operator*(const Vec3& other) const {
    return Vec3(
        other.x * x0 + other.y * y0 + other.z * z0,
        other.x * x1 + other.y * y1 + other.z * z1,
        other.x * x2 + other.y * y2 + other.z * z2);
}

// Matrix-matrix multiplication operator
Mat3 Mat3::operator*(const Mat3& other) const {
    return Mat3(
        x0 * other.x0 + y0 * other.x1 + z0 * other.x2,
        x0 * other.y0 + y0 * other.y1 + z0 * other.y2,
        x0 * other.z0 + y0 * other.z1 + z0 * other.z2,
        x1 * other.x0 + y1 * other.x1 + z1 * other.x2,
        x1 * other.y0 + y1 * other.y1 + z1 * other.y2,
        x1 * other.z0 + y1 * other.z1 + z1 * other.z2,
        x2 * other.x0 + y2 * other.x1 + z2 * other.x2,
        x2 * other.y0 + y2 * other.y1 + z2 * other.y2,
        x2 * other.z0 + y2 * other.z1 + z2 * other.z2);
}

// ----- //
//           Ray           //
// ----- //

// Default constructor
Ray::Ray(Vec3 _origin, Vec3 _direction)
    : origin(_origin), direction(_direction.normalise()) {
}

// Copy constructor
Ray::Ray(const Ray& other)
    : origin(other.origin), direction(other.direction) {
}

// Get the origin of the ray
Vec3 Ray::getOrigin() const {

```

```

        return origin;
    }
    // Get the direction of the ray
    Vec3 Ray::getDirection() const {
        return direction;
    }
    // Set the origin of the ray
    void Ray::setOrigin(const Vec3& _origin) {
        origin = _origin;
    }
    // Set the direction of the ray
    void Ray::setDirection(const Vec3& _direction) {
        direction = _direction;
    }
    // Normalise the direction of the ray
    void Ray::normalise() {
        direction = direction.normalise();
    }
    // Get the point at distance 't' from the origin along the direction vector
    Vec3 Ray::project(const float t) const {
        return origin + direction * t;
    }
}

```

Python

setup.py

```

from distutils.core import setup, Extension, DEBUG

## name          : The listed name of the module
## sources       : The source files to be included in compilation
## include_dirs  : The directories containing necessary header files
## library_dirs  : The directories containing necessary library files
## libraries     : The names of the libraries being used

module = Extension(
    name          = "RayTracer",
    sources       = [
        "module.cpp",
        "BVH.cpp",
        "camera.cpp",

```

```

        "geometry.cpp",
        "model.cpp",
        "modelloader.cpp",
        "transform.cpp"
    ],
    include_dirs = ["C:\\Development\\SDL2\\include"],
    library_dirs = ["C:\\Development\\SDL2\\lib\\x86"],
    libraries     = ["SDL2"]
)

setup(
    name          = "RayTracer",
    version       = "1.0",
    description    = "Renders images",
    ext_modules   = [module]
)

```

main.py

```

from mainwindow import MainWindow

if __name__ == "__main__":
    ## Create main window and start main loop
    mainWin = MainWindow()
    mainWin.mainloop()

```

mainwindow.py

```

import tkinter as tk
import sys

from modeldata import ModelData
from modelwindow import ModelWindow

import RayTracer

class MainWindow(tk.Tk):

    def __init__(self):

```

```

self.modelList = list()
self.validArguments = True
## initialise the tkinter superclass
## that MainWindow inherited from
tk.Tk.__init__(self)
self.configureWindow()
self.setBinds()
self.initLabels()
self.initEntries()
self.initListboxes()
self.initButtons()

## Set all the required window properties
def configureWindow(self):
    self.title("Ray Tracer")
    self.resizable(False, False)

## Bind keyboard inputs to their required functions
def setBinds(self):
    self.bind(
        "<Escape>",
        self.quitGUI)

## Initialise labels, and place them in the window
def initLabels(self):
    tk.Label(self, text="Width:").grid(row=0, column=1, columnspan=3)
    tk.Label(self, text="Height:").grid(row=2, column=1, columnspan=3)
    tk.Label(self, text="Camera FOV (degrees:)").grid(row=4, column=1,
columnspan=3)
    tk.Label(self, text="Camera position (x, y, z:)").grid(row=6,
column=1, columnspan=3)

## Create all entries
def initEntries(self):
    ## Initialise entries
    self.widthEntry = tk.Entry(self, width=12)
    self.heightEntry = tk.Entry(self, width=12)
    self.camFOVEntry = tk.Entry(self, width=12)
    self.camxEntry = tk.Entry(self, width=6)
    self.camyEntry = tk.Entry(self, width=6)
    self.camzEntry = tk.Entry(self, width=6)
    ## Insert default data into the entries

```



```

self.widthEntry.insert(0, "1600")
self.heightEntry.insert(0, "900")
self.camFOVEntry.insert(0, "25.0")
self.camxEntry.insert(0, "0.0")
self.camyEntry.insert(0, "0.0")
self.camzEntry.insert(0, "-10.0")
## Place entries in the window
self.widthEntry.grid(row=1, column=1, columnspan=3)
self.heightEntry.grid(row=3, column=1, columnspan=3)
self.camFOVEntry.grid(row=5, column=1, columnspan=3)
self.camxEntry.grid(row=7, column=1)
self.camyEntry.grid(row=7, column=2)
self.camzEntry.grid(row=7, column=3)

## Create Listboxes
def initListboxes(self):
    ## Initialise listbox
    self.listbox = tk.Listbox(self, width=15)
    ## Get default model data and insert into the listbox
    defaultModel = self.getDefaultModel()
    self.listbox.insert(tk.END, defaultModel.name)
    self.modelList.append(defaultModel)
    ## Bind keyboard inputs to their required functions
    self.listbox.bind(
        "<Double-Button>",
        self.spawnModelWindow)
    self.listbox.bind(
        "<Return>",
        self.spawnModelWindow)
    self.listbox.bind(
        "<Delete>",
        self.deleteModel)
    ## Place listbox in the window
    self.listbox.grid(row=0, column=0, rowspan=6, padx=10)

## Create buttons
def initButtons(self):
    ## Initialise buttons, including the
    ## functions to be called when they are clicked
    self.startButton = tk.Button(self, text="Start",
command=self.start)
    self.exitButton = tk.Button(self, text="Exit",

```

```

command=self.quitGUI)
    self.addButton = tk.Button(self, text="Add", command=self.addModel)
    self.deleteButton = tk.Button(self, text="Delete",
command=self.deleteModel)
    ## Place buttons in the window
    self.startButton.grid(row=8, column=0, padx=10, pady=10,
sticky=tk.E)
    self.exitButton.grid(row=8, column=0, padx=10, pady=10,
sticky=tk.W)
    self.addButton.grid(row=6, column=0, padx=10, pady=10, sticky=tk.E)
    self.deleteButton.grid(row=6, column=0, padx=10, pady=10,
sticky=tk.W)

    ## Start the renderer if inputs are valid
    def start(self):
        self.validate()
        if self.validArguments:
            self.initRender()

    ## Validate data in the entries
    def validate(self):
        attrbValidityDict = self.getValidation()
        self.highlightInvalidEntries(attrbValidityDict)
        if False in attrbValidityDict.values():
            self.validArguments = False
        else:
            self.validArguments = True

    ## Create dictionary that stores whether attributes are valid or not
    def createAttrbValidityDict(self):
        attrbValidityDict = {
            "camx":True, "camy":True, "camz":True,
            "camfov":True, "width":True, "height":True
        }
        return attrbValidityDict

    ## Check for validity of each entry,
    ## storing results in a validity dictionary
    def getValidation(self):
        attrbValidityDict = self.createAttrbValidityDict()
        if not self.widthEntry.get().isdigit():
            attrbValidityDict["width"] = False

```

```

if not self.heightEntry.get().isdigit():
    attrbValidityDict["height"] = False
if not self.validFloat(self.camxEntry.get()):
    attrbValidityDict["camx"] = False
if not self.validFloat(self.camyEntry.get()):
    attrbValidityDict["camy"] = False
if not self.validFloat(self.camzEntry.get()):
    attrbValidityDict["camz"] = False
if not self.validFloat(self.camFOVEntry.get()):
    attrbValidityDict["camfov"] = False
return attrbValidityDict

## Validate whether a string could be
## converted to a valid floating point number
def validFloat(self, numString):
    try:
        float(numString)
    except ValueError:
        return False
    return True

## For each entry, if data is valid
## highlight red, if invalid highlight red
def highlightInvalidEntries(self, attrbValidityDict):
    ## Cache the wierd dictionary arguments
    ## that Tkinter uses to configure entries
    redBG = {"background": "Red"}
    whiteBG = {"background": "White"}
    if not attrbValidityDict["width"]:
        self.widthEntry.configure(redBG)
    else:
        self.widthEntry.configure(whiteBG)
    if not attrbValidityDict["height"]:
        self.heightEntry.configure(redBG)
    else:
        self.heightEntry.configure(whiteBG)
    if not attrbValidityDict["camx"]:
        self.camxEntry.configure(redBG)
    else:
        self.camxEntry.configure(whiteBG)
    if not attrbValidityDict["camy"]:
        self.camyEntry.configure(redBG)

```

```

else:
    self.camyEntry.configure(whiteBG)
if not attrbValidityDict["camz"]:
    self.camzEntry.configure(redBG)
else:
    self.camzEntry.configure(whiteBG)
if not attrbValidityDict["camfov"]:
    self.camFOVEntry.configure(redBG)
else:
    self.camFOVEntry.configure(whiteBG)

## Create a ModelWindow for the selected model
def spawnModelWindow(self, event):
    ## Get the index of the selected model in the listbox
    index = int(event.widget.curselection()[0])
    modelWin = ModelWindow(self, self.modelList[index])

## Fully quit the program
def quitGUI(self, event=None):
    ## Quit Tkinter
    self.destroy()
    ## Quit Python
    sys.exit()

## Delete the selected models
def deleteModel(self):
    ## Get indices of selected models
    indices = self.listbox.curselection()
    ## Delete the models iterating backwards
    ## through the list, otherwise when the list
    ## shifts all elements left after the deleted
    ## element forwards, it would screw with the
    ## indices of the models still to be deleted
    for index in reversed(indices):
        model = self.modelList[index]
        ## Remove the model both from the
        ## listbox, and the internal list
        self.modelList.remove(model)
        self.listbox.delete(index)

## Get default configuration for new models
def getDefaultModel(self):

```

```

        return ModelData(
            "sphere.obj",
            0.0, 0.0, 0.0,
            0.0, 0.0, 0.0,
            False, False, False)

    ## Add the default model, and spawn
    ## a ModelWindow to configure it
    def addModel(self):
        index = len(self.modelList)
        defaultModel = self.getDefaultModel()
        self.listbox.insert(tk.END, defaultModel.name)
        self.modelList.append(defaultModel)
        modelWin = ModelWindow(self, self.modelList[index])

    ## Update the listed name of a given model in the listbox
    def updateModelName(self, model):
        index = self.modelList.index(model)
        self.listbox.delete(index)
        self.listbox.insert(index, model.name)

    ## Pack window and model attributes
    ## to be passed to the C++ program
    def packArguments(self):
        ## Pack non-model attributes
        args = [
            self.getWidth(),
            self.getHeight(),
            self.getCamx(),
            self.getCamy(),
            self.getCamz(),
            self.getCamFOV()
        ]
        ## For each model, add the model's attributes to the list
        for model in self.modelList:
            args += model.packArguments()
        ## Convert the list to a tuple, and return.
        ## For whatever reason, C++ extensions
        ## only accept tuple packed arguments
        return tuple(args)

    ## Start the renderer with the required arguments

```

```

def initRender(self):
    RayTracer.render(
        self.packArguments()
    )

def getWidth(self):
    return int(self.widthEntry.get())

def getHeight(self):
    return int(self.heightEntry.get())

def getCamx(self):
    return float(self.camxEntry.get())

def getCamy(self):
    return float(self.camyEntry.get())

def getCamz(self):
    return float(self.camzEntry.get())

def getCamFOV(self):
    return float(self.camFOVEntry.get())

```

modelwindow.py

```

import tkinter as tk
from modeldata import ModelData

class ModelWindow(tk.Toplevel):

    def __init__(self, parent, model):
        self.parent = parent
        self.model = model
        self.validModelArguments = True
        ## initialise the tkinter superclass
        ## that MainWindow inherited from
        tk.Toplevel.__init__(self)
        self.configureWindow(model.name)
        self.setBinds()
        self.initLabels()
        self.initEntries()

```

```

        self.initButtons()

## Set all the required window properties
def configureWindow(self, modelName):
    self.focus_force()
    self.title(modelName)
    self.resizable(False, False)
    self.grab_set()

## Bind keyboard inputs to their required functions
def setBinds(self):
    self.bind(
        "<Escape>",
        self.destroyWindow)

## Initialise labels, and place them in the window
def initLabels(self):
    tk.Label(self, text="File name (model.obj):").grid(row=0, column=0,
padx=10, pady=10)
    tk.Label(self, text="Position (x, y, z):").grid(row=1, column=0,
padx=10, pady=10)
    tk.Label(self, text="Rotation (degrees):").grid(row=2, column=0,
padx=10, pady=10)
    tk.Label(self, text="Flip (x, y, z):").grid(row=3, column=0,
padx=10, pady=10)

## Create all entries
def initEntries(self):
    self.initFilenameEntry()
    self.initPosEntries()
    self.initRotEntries()
    self.initFlipChecks()

## Create filename entries
def initFilenameEntry(self):
    ## Initialise entry
    self.filenameEntry = tk.Entry(self)
    ## Insert model data into entry
    self.filenameEntry.insert(0, self.model.filename)
    ## Place entry in the window
    self.filenameEntry.grid(row=0, column=1, columnspan=3, padx=10,
pady=10)

```

```

## Create position entries
def initPosEntries(self):
    ## Initialise entries
    self.xEntry = tk.Entry(self, width=5)
    self.yEntry = tk.Entry(self, width=5)
    self.zEntry = tk.Entry(self, width=5)
    ## Insert model data into entries
    self.xEntry.insert(0, str(self.model.x))
    self.yEntry.insert(0, str(self.model.y))
    self.zEntry.insert(0, str(self.model.z))
    ## Place entries in the window
    self.xEntry.grid(row=1, column=1, padx=10, pady=10)
    self.yEntry.grid(row=1, column=2, padx=10, pady=10)
    self.zEntry.grid(row=1, column=3, padx=10, pady=10)

## Create rotation entries
def initRotEntries(self):
    ## Initialise entries
    self.rotxEntry = tk.Entry(self, width=5)
    self.rotyEntry = tk.Entry(self, width=5)
    self.rotzEntry = tk.Entry(self, width=5)
    ## Insert model data into entries
    self.rotxEntry.insert(0, str(self.model.rotx))
    self.rotyEntry.insert(0, str(self.model.roty))
    self.rotzEntry.insert(0, str(self.model.rotz))
    ## Place entries in the window
    self.rotxEntry.grid(row=2, column=1, padx=10, pady=10)
    self.rotyEntry.grid(row=2, column=2, padx=10, pady=10)
    self.rotzEntry.grid(row=2, column=3, padx=10, pady=10)

## Create flip check buttons
def initFlipChecks(self):
    ## Initialise integer variables used for
    ## storing the state of the checkbuttons
    vx = tk.IntVar()
    vy = tk.IntVar()
    vz = tk.IntVar()
    ## Initialise check buttons
    self.flipxCheck = tk.Checkbutton(self, variable=vx)
    self.flipyCheck = tk.Checkbutton(self, variable=vy)
    self.flipzCheck = tk.Checkbutton(self, variable=vz)

```



```

    ## Set check button variables as the integer variables
    self.flipxCheck.var = vx
    self.flipyCheck.var = vy
    self.flipzCheck.var = vz
    ## Insert model data into check buttons
    if self.model.flipx:
        self.flipxCheck.toggle()
    if self.model.flipy:
        self.flipyCheck.toggle()
    if self.model.flipz:
        self.flipzCheck.toggle()
    ## Place check buttons in the window
    self.flipxCheck.grid(row=3, column=1, padx=10, pady=10)
    self.flipyCheck.grid(row=3, column=2, padx=10, pady=10)
    self.flipzCheck.grid(row=3, column=3, padx=10, pady=10)

    ## Create buttons
    def initButtons(self):
        ## Initialise buttons, including the
        ## functions to be called when they are clicked
        self.cancelButton = tk.Button(self, text="Cancel",
command=self.destroyWindow)
        self.acceptButton = tk.Button(self, text="Accept",
command=self.windowExit)
        ## Place buttons in the window
        self.cancelButton.grid(row=4, column=0, padx=10, pady=10,
sticky=tk.W)
        self.acceptButton.grid(row=4, column=3, padx=10, pady=10,
sticky=tk.E)

    ## return to main window if inputs are valid
    def windowExit(self):
        self.saveAndValidate()
        if self.validModelArguments:
            self.parent.updateModelName(self.model)
            self.destroyWindow()

    ## Save and validate the input data
    def saveAndValidate(self):
        self.saveModelData()
        self.validateModelData()

```

```

## Save entry and check button contents to the model
def saveModelData(self):
    self.model.filename = self.filenameEntry.get()
    self.model.name = (self.model.filename.split(".")[0]).capitalize()
    self.model.x = self.xEntry.get()
    self.model.y = self.yEntry.get()
    self.model.z = self.zEntry.get()
    self.model.rotx = self.rotxEntry.get()
    self.model.rotz = self.rotzEntry.get()
    self.model.flipx = self.flipxCheck.var.get()
    self.model.flipy = self.flipyCheck.var.get()
    self.model.flipz = self.flipzCheck.var.get()

## Check whether the input data is valid
def validateModelData(self):
    attrbValidityDict = self.model.getValidation()
    self.highlightInvalidEntries(attrbValidityDict)
    if False in attrbValidityDict.values():
        self.validModelArguments = False
    else:
        self.validModelArguments = True

## For each entry, if data is valid
## highlight red, if invalid highlight red
def highlightInvalidEntries(self, attrbValidityDict):
    redBG = {"background": "Red"}
    whiteBG = {"background": "White"}
    if not attrbValidityDict["filename"]:
        self.filenameEntry.configure(redBG)
    else:
        self.filenameEntry.configure(whiteBG)

    self.highlightInvalidPosEntries(attrbValidityDict)
    self.highlightInvalidRotEntries(attrbValidityDict)

## Highlight position entries
def highlightInvalidPosEntries(self, attrbValidityDict):
    redBG = {"background": "Red"}
    whiteBG = {"background": "White"}
    if not attrbValidityDict["x"]:
        self.xEntry.configure(redBG)

```

```

else:
    self.xEntry.configure(whiteBG)
if not attrbValidityDict["y"]:
    self.yEntry.configure(redBG)
else:
    self.yEntry.configure(whiteBG)
if not attrbValidityDict["z"]:
    self.zEntry.configure(redBG)
else:
    self.zEntry.configure(whiteBG)

## Highlight rotation entries
def highlightInvalidRotEntries(self, attrbValidityDict):
    redBG = {"background": "Red"}
    whiteBG = {"background": "White"}
    if not attrbValidityDict["rotx"]:
        self.rotxEntry.configure(redBG)
    else:
        self.rotxEntry.configure(whiteBG)
    if not attrbValidityDict["roty"]:
        self.rotyEntry.configure(redBG)
    else:
        self.rotyEntry.configure(whiteBG)
    if not attrbValidityDict["rotz"]:
        self.rotzEntry.configure(redBG)
    else:
        self.rotzEntry.configure(whiteBG)

## Quit the window, returning
## control to the main window
def destroyWindow(self, event=None):
    self.grab_release()
    self.destroy()

```

modeldata.py

```

class ModelData:

    def __init__(self, filename,
                  x, y, z,
                  rotx, roty, rotz,

```

```

        flipx, flipy, flipz):
self.filename = filename
self.x = x
self.y = y
self.z = z
self.rotx = rotx
self.roty = roty
self.rotz = rotz
self.flipx = flipx
self.flipy = flipy
self.flipz = flipz
self.name = (filename.split(".")[0]).capitalize()

## Create dictionary that stores whether attributes are valid or not
def createAttrbValidityDict(self):
    attrbValidityDict = {
        "x":True, "y":True, "z":True,
        "rotx":True, "roty":True, "rotz":True,
        "filename":True
    }
    return attrbValidityDict

## Check for validity of each attribute,
## storing results in a validity dictionary
def getValidation(self):
    attrbValidityDict = self.createAttrbValidityDict()
    if not self.validFilename(self.filename):
        attrbValidityDict["filename"] = False
    if not self.validFloat(self.x):
        attrbValidityDict["x"] = False
    if not self.validFloat(self.y):
        attrbValidityDict["y"] = False
    if not self.validFloat(self.z):
        attrbValidityDict["z"] = False
    if not self.validFloat(self.rotx):
        attrbValidityDict["rotx"] = False
    if not self.validFloat(self.roty):
        attrbValidityDict["roty"] = False
    if not self.validFloat(self.rotz):
        attrbValidityDict["rotz"] = False
    return attrbValidityDict

```

```

## Check whether the given filename is syntactically
## valid (not whether it exists on the computer)
def validFilename(self, filename):
    if not filename.lower().endswith(".obj"):
        return False
    elif not len(filename.split(".")) == 2:
        return False
    return True

## Validate whether a string could be
## converted to a valid floating point number
def validFloat(self, numString):
    try:
        float(numString)
    except ValueError:
        return False
    return True

## Pack attributes to be passed to the C++ program
def packArguments(self):
    return [
        self.filename,
        self.x,
        self.y,
        self.z,
        self.rotx,
        self.roty,
        self.rotz,
        self.flipx,
        self.flipy,
        self.flipz
    ]

```

Testing

Video demonstration

<https://www.youtube.com/watch?v=qTfYlJooVyk>

The rendering is done very quickly as it is being run on a powerful processor (Intel Xeon E3-1230) with 16GB of RAM available.

Renderer testing

Test No	Purpose of test	Description of test	Test data	Expected result	Actual result	Image references
1	To test if vertices are read into the lists correctly from the OBJ file, in order.	Print the vertices as they're read from the file, and compare to the OBJ file itself	'cube.obj', 'sphere.obj'	Printed vertices are the same as those in the file	As expected	1, 2
2	To test if normals are read into the lists correctly from the OBJ file, in order.	Print the normals as they're read from the file, and compare to the OBJ file itself	'cube.obj', 'sphere.obj'	Printed normals are the same as those in the file	As expected	3, 4
3	To test if vertex and normal indices are read into the 'Triangle' objects correctly from the OBJ file.	Print the indices as they're read from the file, and compare to the OBJ file itself	'cube.obj', 'sphere.obj'	Printed indices are the same as those in the file	As expected	5, 6
4	To test if the BVH stores the vertex and normal	Print the vertex and normal indices as they're	'cube.obj', 'sphere.obj'	Printed indices are the same as those in the file,	As expected	7, 8

	indices correctly	iterated through in the BVH, and compare to the OBJ file itself		minus 1 (Indices in the OBJ file start at 1, so they must be decremented to start at 0)		
5	To test if the model stores the vertices correctly	Print the vertices as they're iterated through in the model, and compare to the OBJ file itself	'cube.obj', 'sphere.obj'	Printed vertices are the same as those in the file	As expected	9, 10
6	To test if the model stores the normals correctly	Print the normals as they're iterated through in the model, and compare to the OBJ file itself	'cube.obj', 'sphere.obj'	Printed normals are the same as those in the file	As expected	11, 12
7	To test if the BVH splits the list of triangles into evenly sized subsets	After each partition print the size of each of the two subsets	'cube.obj', 'sphere.obj'	Lengths of each subset are somewhat similar	As expected	13, 14
8	To test if rays are emitted from the camera's	Iterate through the emitted rays, if any	Set camera origin to: (0, 0, -10), (1, 4, 42), (0, 0, 0)	There are no differences	As expected	15, 16, 17

	origin	of the ray origins differ from the camera origin, print a flag in the console				
9	To test if the ray recursively checks for intersections with a nodes children if it intersects the parent node	Print the depth of each node that is intersected with	'cube.obj', 'sphere.obj'	A typical pre-order binary tree search	As expected	18, 19
10	To test if, when a leaf node is intersected with, the ray iterates through the triangles of the node and tests for intersections with them	Print when a leaf node is intersected with, and then when each triangle of the node is tested against	'cube.obj', 'sphere.obj'	After each leaf node intersection, a number of triangles are tested against	As expected	20, 21
11	To test if the distance to the intersection point t, between the the	Print the value of t returned for each ray	'cube.obj', 'sphere.obj'	Values of t that are approximately the expected distance between the camera	As expected	22, 23

	triangle and the ray is returned			origin and the object's surface		
12	To test if, if multiple triangles overlap, the closest one is rendered for a particular pixel	Draw an object with multiple 'layers' of geometry, to make sure that each layer is drawn in the correct order	'car.obj'	The model is drawn correctly, not 'inside out'		24
13	To test if back facing triangles are ignored from rendering	For each triangle that is intersected with, print whether it is back facing	'cube.obj', 'sphere.obj'	A number of triangles will be identified as back facing, and ignored	As expected	25, 26
14	To test if the pixel is set to the background colour if no triangles are intersected with	Set the background colour, and see what the background is in the final image	Set background colour to: (0.02, 0.02, 0.04), then, (0.5, 0.3, 1.0)	The background will be a dark grey, then it will be a light, desaturated purple	As expected	27, 28
15	To test if triangles are rendered with directional, flat shading	Draw objects, and check if the triangles are shaded as described	'sphere.obj', 'car.obj'	The triangles are shaded, getting brighter as they angle towards the light source	As expected	29, 30

16	To test if models are rendered in the correct order, with models closer to the camera rendered in front of those further away	Draw two overlapping objects, check that they're correctly ordered	cube in front of sphere, sphere in front of cube, sphere and cube at similar distances to the camera	cube in front of sphere, sphere in front of cube, sphere and cube intersecting	As expected	31, 32, 33
----	---	--	--	--	-------------	------------

GUI testing

Main

Test No	Purpose of test	Description of test	Test data	Expected result	Actual result	Image references
1	To test whether the GUI is launched from the entry script	The main script is run by double clicking on the 'main.py' file in windows explorer	No test data	The GUI window opens	The GUI window opens	1

MainWindow

Test No	Purpose of test	Description of test	Test data	Expected result	Actual result	Image references
2	To test whether invalid inputs are	Enter invalid data into the inputs,	Set only width to "blah",	The invalid entries are highlighted red, the	As expected	2, 3, 4

	recognised and highlighted red.	and press start.	Set both height and camera FOV to "-1", Set camera position to (1, 1, "aba")	renderer does not start.		
3	To test whether inputs that were previously invalid and now valid are returned to having a white background.	After entering invalid data, change some of the entries to valid data and try to start the program. Make sure there is at least one invalid entry left so that the program doesn't start.	Set width and height to -20, and attempt to start to highlight them both red, then change height to 50 and attempt to start again	The newly valid entries return to a white background, the invalid entries remain red.	As expected	5, 6
4	To test whether arguments are correctly packed into the list sent to the C++ application	Print the list that is returned by 'packArguments' and inspect it for errors	Test data shown in screenshots	The data in the list is consistent with that specified by the user in the GUI	As expected	7, 8, 9, 10

ModelWindow

Test No	Purpose of test	Description of test	Test data	Expected result	Actual result	Image references
4	To test whether invalid inputs are recognised and highlighted red.	Enter invalid data into the inputs, and press start.	Set filename to "sphere", then "obj.sphere", then "sphere.png" Set position to (-, ~, 2)	The invalid entries are highlighted red, the renderer does not start.	As expected	11, 12, 13, 14
5	To test whether inputs that were previously invalid and now valid are returned to having a white background.	After entering invalid data, change some of the entries to valid data and try to start the program. Make sure there is at least one invalid entry left so that the program doesn't start.	Set filename to "152" and rotation to (a, b, c), and press accept to highlight the invalid entries red, Then change filename to "mill.obj" and rotation to (0.0, b, c) and press accept	The newly valid entries return to a white background, the invalid entries remain red.	As expected	15, 16

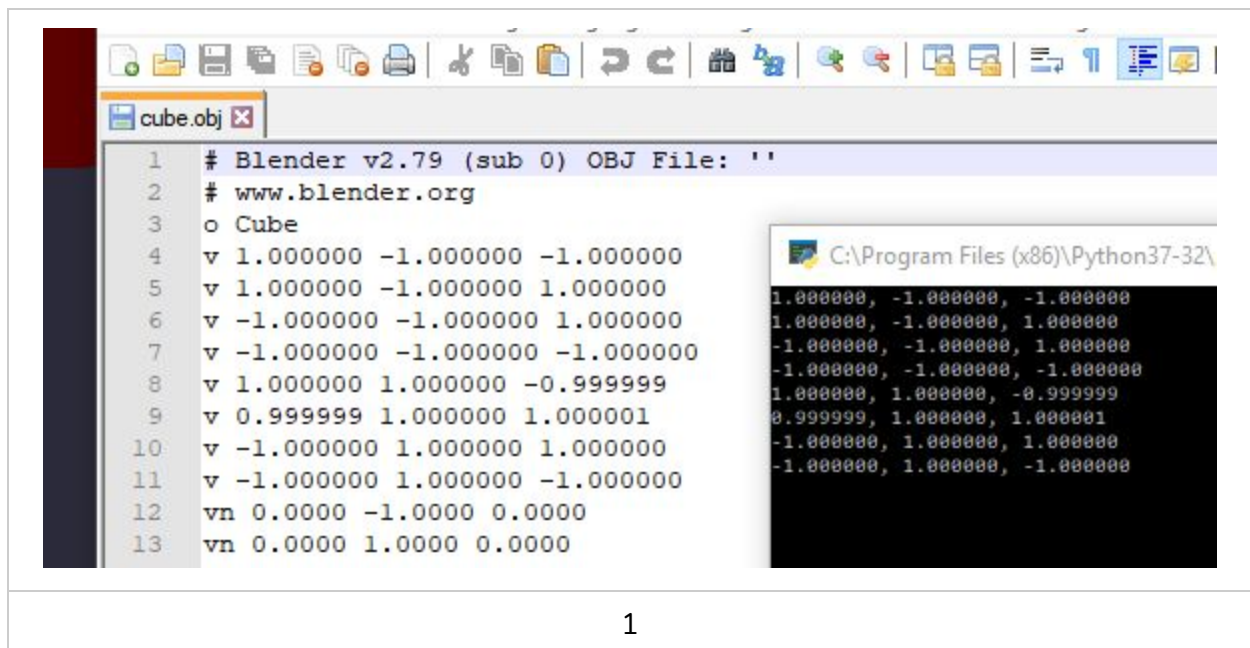
ModelData

Test No	Purpose of test	Description of test	Test data	Expected result	Actual result	Image references
6	To test	Print the	Filename =	The data in	As	17, 18

	whether the arguments are correctly packed into the list	list that is returned by 'packArguments' and inspect it for errors	"car.obj", position = (4, 0, -2), rotation = (30, 2, -6), flip = (false, true, true)	the list is consistent with that specified by the user in the GUI	expected	
--	--	--	--	---	----------	--

Testing images

Renderer



```
sphere.obj
1 # Blender v2.79 (sub 0) OBJ File: ''
2 # www.blender.org
3 o Icosphere
4 v 0.000000 -1.000000 0.000000
5 v 0.723607 -0.447220 0.525725
6 v -0.276388 -0.447220 0.850649
7 v -0.894426 -0.447216 0.000000
8 v -0.276388 -0.447220 -0.850649
9 v 0.723607 -0.447220 -0.525725
10 v 0.276388 0.447220 0.850649
11 v -0.723607 0.447220 0.525725
12 v -0.723607 0.447220 -0.525725
13 v 0.276388 0.447220 -0.850649
14 v 0.894426 0.447216 0.000000
15 v 0.000000 1.000000 0.000000
16 v -0.162456 -0.850654 0.499995
17 v 0.425323 -0.850654 0.309011
18 v 0.262869 -0.525738 0.809012
19 v 0.850648 -0.525736 0.000000
20 v 0.425323 -0.850654 -0.309011
21 v -0.525730 -0.850652 0.000000
22 v -0.688189 -0.525736 0.499997
23 v -0.162456 -0.850654 -0.499995
24 v -0.688189 -0.525736 -0.499997
25 v 0.262869 -0.525738 -0.809012
26 v 0.951058 0.000000 0.309013
27 v 0.951058 0.000000 -0.309013
28 v 0.000000 0.000000 1.000000
29 v 0.587786 0.000000 0.809017
30 v -0.951058 0.000000 0.309013
31 v -0.587786 0.000000 0.809017
32 v -0.587786 0.000000 -0.809017
33 v -0.951058 0.000000 -0.309013
34 v 0.587786 0.000000 -0.809017
35 v 0.000000 0.000000 -1.000000
36 v 0.688189 0.525736 0.499997
37 v -0.262869 0.525738 0.809012
38 v -0.850648 0.525736 0.000000
39 v -0.262869 0.525738 -0.809012
40 v 0.688189 0.525736 -0.499997
41 v 0.162456 0.850654 0.499995
42 v 0.525730 0.850652 0.000000
43 v -0.425323 0.850654 0.309011
44 v -0.425323 0.850654 -0.309011
45 v 0.162456 0.850654 -0.499995
46 vn 0.1024 -0.9435 0.3151
```

```

10 v -1.000000 1.000000 1.000000
11 v -1.000000 1.000000 -1.000000
12 vn 0.0000 -1.0000 0.0000
13 vn 0.0000 1.0000 0.0000
14 vn 1.0000 -0.0000 0.0000
15 vn 0.0000 -0.0000 1.0000
16 vn -1.0000 -0.0000 -0.0000
17 vn 0.0000 0.0000 -1.0000
18 s off
19 # 0/0 1/0 1/0

```

```

0.000000, -1.000000, 0.000000
0.000000, 1.000000, 0.000000
1.000000, -0.000000, 0.000000
0.000000, -0.000000, 1.000000
-1.000000, -0.000000, -0.000000
0.000000, 0.000000, -1.000000

```


sphere.obj

```
46 vn 0.1024 -0.9435 0.3151
47 vn 0.7002 -0.6617 0.2680
48 vn -0.2680 -0.9435 0.1947
49 vn -0.2680 -0.9435 -0.1947
50 vn 0.1024 -0.9435 -0.3151
51 vn 0.9050 -0.3304 0.2680
52 vn 0.0247 -0.3304 0.9435
53 vn -0.8897 -0.3304 0.3151
54 vn -0.5746 -0.3304 -0.7488
55 vn 0.5346 -0.3304 -0.7779
56 vn 0.8026 -0.1256 0.5831
57 vn -0.3066 -0.1256 0.9435
58 vn -0.9921 -0.1256 0.0000
59 vn -0.3066 -0.1256 -0.9435
60 vn 0.8026 -0.1256 -0.5831
61 vn 0.4089 0.6617 0.6284
62 vn -0.4713 0.6617 0.5831
63 vn -0.7002 0.6617 -0.2680
64 vn 0.0385 0.6617 -0.7488
65 vn 0.7240 0.6617 -0.1947
66 vn 0.2680 0.9435 -0.1947
67 vn 0.4911 0.7947 -0.3568
68 vn 0.4089 0.6617 -0.6284
69 vn -0.1024 0.9435 -0.3151
70 vn -0.1876 0.7947 -0.5773
71 vn -0.4713 0.6617 -0.5831
72 vn -0.3313 0.9435 0.0000
73 vn -0.6071 0.7947 0.0000
74 vn -0.7002 0.6617 0.2680
75 vn -0.1024 0.9435 0.3151
76 vn -0.1876 0.7947 0.5773
77 vn 0.0385 0.6617 0.7488
78 vn 0.2680 0.9435 0.1947
79 vn 0.4911 0.7947 0.3568
80 vn 0.7240 0.6617 0.1947
81 vn 0.8897 0.3304 -0.3151
82 vn 0.7947 0.1876 -0.5773
83 vn 0.5746 0.3304 -0.7488
84 vn -0.0247 0.3304 -0.9435
85 vn -0.3035 0.1876 -0.9342
86 vn -0.5346 0.3304 -0.7779
87 vn -0.9050 0.3304 -0.2680
88 vn -0.9822 0.1876 0.0000
89 vn -0.9050 0.3304 0.2680
90 vn -0.5346 0.3304 0.7779
91 vn -0.3035 0.1876 0.9342
92 vn -0.0247 0.3304 0.9435
93 vn 0.5746 0.3304 0.7488
```

C:\Program Files (x86)\Python37-32

```
0.102400, -0.943500, 0.315100
0.700200, -0.661700, 0.268000
-0.268000, -0.943500, 0.194700
-0.268000, -0.943500, -0.194700
0.102400, -0.943500, -0.315100
0.905000, -0.330400, 0.268000
0.024700, -0.330400, 0.943500
-0.889700, -0.330400, 0.315100
-0.574600, -0.330400, -0.748800
0.534600, -0.330400, -0.777900
0.802600, -0.125600, 0.583100
-0.306600, -0.125600, 0.943500
-0.992100, -0.125600, 0.000000
-0.306600, -0.125600, -0.943500
0.802600, -0.125600, -0.583100
0.408900, 0.661700, 0.628400
-0.471300, 0.661700, 0.583100
-0.700200, 0.661700, -0.268000
0.038500, 0.661700, -0.748800
0.724000, 0.661700, -0.194700
0.268000, 0.943500, -0.194700
0.491100, 0.794700, -0.356800
0.408900, 0.661700, -0.628400
-0.102400, 0.943500, -0.315100
-0.187600, 0.794700, -0.577300
-0.471300, 0.661700, -0.583100
-0.331300, 0.943500, 0.000000
-0.607100, 0.794700, 0.000000
-0.700200, 0.661700, 0.268000
-0.102400, 0.943500, 0.315100
-0.187600, 0.794700, 0.577300
0.038500, 0.661700, 0.748800
0.268000, 0.943500, 0.194700
0.491100, 0.794700, 0.356800
0.724000, 0.661700, 0.194700
0.889700, 0.330400, -0.315100
0.794700, 0.187600, -0.577300
0.574600, 0.330400, -0.748800
-0.024700, 0.330400, -0.943500
-0.303500, 0.187600, -0.934200
-0.534600, 0.330400, -0.777900
-0.905000, 0.330400, -0.268000
-0.982200, 0.187600, 0.000000
-0.905000, 0.330400, 0.268000
-0.534600, 0.330400, 0.777900
-0.303500, 0.187600, 0.934200
-0.024700, 0.330400, 0.943500
```



```

18 s off
19 f 2//1 4//1 1//1 v: 2, 4, 1
20 f 8//2 6//2 5//2 n: 1, 1, 1
21 f 5//3 2//3 1//3 v: 8, 6, 5
22 f 6//4 3//4 2//4 n: 2, 2, 2
23 f 3//5 8//5 4//5 v: 5, 2, 1
24 f 1//6 8//6 5//6 n: 3, 3, 3
25 f 2//1 3//1 4//1 v: 6, 3, 2
26 f 8//2 7//2 6//2 n: 4, 4, 4
27 f 5//3 6//3 2//3 v: 3, 8, 4
28 f 6//4 7//4 3//4 n: 5, 5, 5
29 f 3//5 7//5 8//5 v: 1, 8, 5
30 f 1//6 4//6 8//6 n: 6, 6, 6
31

```

124	vn 0.1876 -0.7947 0.577	
125	vn 0.4713 -0.6617 0.583	
126	s off	
127	f 1//1 14//1 13//1	v: 1, 14, 13 n: 1, 1, 1
128	f 2//2 14//2 16//2	v: 2, 14, 16 n: 2, 2, 2
129	f 1//3 13//3 18//3	
130	f 1//4 18//4 20//4	v: 1, 13, 18 n: 3, 3, 3
131	f 1//5 20//5 17//5	
132	f 2//6 16//6 23//6	v: 1, 18, 20 n: 4, 4, 4
133	f 3//7 15//7 25//7	
134	f 4//8 19//8 27//8	v: 1, 20, 17 n: 5, 5, 5
135	f 5//9 21//9 29//9	
136	f 6//10 22//10 31//10	v: 2, 16, 23 n: 6, 6, 6
137	f 2//11 23//11 26//11	
138	f 3//12 25//12 28//12	v: 3, 15, 25 n: 7, 7, 7
139	f 4//13 27//13 30//13	
140	f 5//14 29//14 32//14	v: 4, 19, 27 n: 8, 8, 8
141	f 6//15 31//15 24//15	
142	f 7//16 33//16 38//16	v: 5, 21, 29 n: 9, 9, 9
143	f 8//17 34//17 40//17	
144	f 9//18 35//18 41//18	v: 6, 22, 31 n: 10, 10, 10
145	f 10//19 36//19 42//19	
146	f 11//20 37//20 39//20	v: 2, 23, 26 n: 11, 11, 11
147	f 39//21 42//21 12//21	
148	f 39//22 37//22 42//22	v: 3, 25, 28 n: 12, 12, 12
149	f 37//23 10//23 42//23	
150	f 42//24 41//24 12//24	v: 4, 27, 30 n: 13, 13, 13
151	f 42//25 36//25 41//25	
152	f 36//26 9//26 41//26	v: 5, 29, 32 n: 14, 14, 14
153	f 41//27 40//27 12//27	
154	f 41//28 35//28 40//28	v: 6, 31, 24 n: 15, 15, 15
155	f 35//29 8//29 40//29	
156	f 40//30 38//30 12//30	v: 7, 33, 38 n: 16, 16, 16
157	f 40//31 34//31 38//31	
158	f 34//32 7//32 38//32	v: 8, 34, 40 n: 17, 17, 17
159	f 38//33 39//33 12//33	
160	f 38//34 33//34 39//34	v: 9, 35, 41 n: 18, 18, 18
161	f 33//35 11//35 39//35	
162	f 24//36 37//36 11//36	v: 10, 36, 42 n: 19, 19, 19
163	f 24//37 31//37 37//37	
164	f 31//38 10//38 37//38	v: 11, 37, 39 n: 20, 20, 20
165	f 32//39 36//39 10//39	
166	f 32//40 29//40 36//40	v: 39, 42, 12 n: 21, 21, 21
167	f 29//41 9//41 36//41	
168	f 30//42 35//42 9//42	
169	f 30//43 27//43 35//43	
170	f 27//44 8//44 35//44	
171	f 28//45 34//45 8//45	
172	f 28//46 25//46 34//46	
173	f 25//47 7//47 34//47	

```
v: 1, 3, 0
n: 0, 0, 0

v: 7, 5, 4
n: 1, 1, 1

v: 4, 1, 0
n: 2, 2, 2

v: 5, 2, 1
n: 3, 3, 3

v: 2, 7, 3
n: 4, 4, 4

v: 0, 7, 4
n: 5, 5, 5

v: 1, 2, 3
n: 0, 0, 0

v: 7, 6, 5
n: 1, 1, 1

v: 4, 5, 1
n: 2, 2, 2

v: 5, 6, 2
n: 3, 3, 3

v: 2, 6, 7
n: 4, 4, 4

v: 0, 3, 7
n: 5, 5, 5
```

126	s off	v: 0, 13, 12
127	f 1//1 14//1 13//1	n: 0, 0, 0
128	f 2//2 14//2 16//2	
129	f 1//3 13//3 18//3	v: 1, 13, 15
130	f 1//4 18//4 20//4	n: 1, 1, 1
131	f 1//5 20//5 17//5	v: 0, 12, 17
132	f 2//6 16//6 23//6	n: 2, 2, 2
133	f 3//7 15//7 25//7	v: 0, 17, 19
134	f 4//8 19//8 27//8	n: 3, 3, 3
135	f 5//9 21//9 29//9	v: 0, 19, 16
136	f 6//10 22//10 31//10	n: 4, 4, 4
137	f 2//11 23//11 26//11	v: 1, 15, 22
138	f 3//12 25//12 28//12	n: 5, 5, 5
139	f 4//13 27//13 30//13	
140	f 5//14 29//14 32//14	v: 2, 14, 24
141	f 6//15 31//15 24//15	n: 6, 6, 6
142	f 7//16 33//16 38//16	v: 3, 18, 26
143	f 8//17 34//17 40//17	n: 7, 7, 7
144	f 9//18 35//18 41//18	v: 4, 20, 28
145	f 10//19 36//19 42//19	n: 8, 8, 8
146	f 11//20 37//20 39//20	v: 5, 21, 30
147	f 39//21 42//21 12//21	n: 9, 9, 9
148	f 39//22 37//22 42//22	
149	f 37//23 10//23 42//23	v: 1, 22, 25
150	f 42//24 41//24 12//24	n: 10, 10, 10
151	f 42//25 36//25 41//25	v: 2, 24, 27
152	f 36//26 9//26 41//26	n: 11, 11, 11
153	f 41//27 40//27 12//27	v: 3, 26, 29
154	f 41//28 35//28 40//28	n: 12, 12, 12
155	f 35//29 8//29 40//29	
156	f 40//30 38//30 12//30	v: 4, 28, 31
157	f 40//31 34//31 38//31	n: 13, 13, 13
158	f 34//32 7//32 38//32	v: 5, 30, 23
159	f 38//33 39//33 12//33	n: 14, 14, 14
160	f 38//34 33//34 39//34	v: 6, 32, 37
161	f 33//35 11//35 39//35	n: 15, 15, 15
162	f 24//36 37//36 11//36	v: 7, 33, 39
163	f 24//37 31//37 37//37	n: 16, 16, 16
164	f 31//38 10//38 37//38	
165	f 32//39 36//39 10//39	v: 8, 34, 40
166	f 32//40 29//40 36//40	n: 17, 17, 17
167	f 29//41 9//41 36//41	v: 9, 35, 41
168	f 30//42 35//42 9//42	n: 18, 18, 18
169	f 30//43 27//43 35//43	v: 10, 36, 38
170	f 27//44 8//44 35//44	n: 19, 19, 19
171	f 28//45 34//45 8//45	v: 38, 41, 11
172	f 28//46 25//46 34//46	n: 20, 20, 20
173	f 25//47 7//47 34//47	
174	f 26//48 33//48 7//48	v: 38, 36, 41
		n: 21, 21, 21

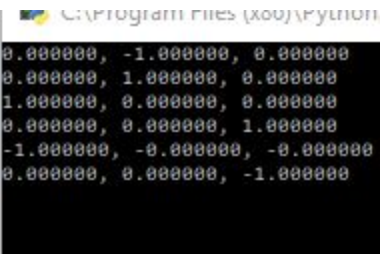
3	o cube	
4	v 1.000000 -1.000000 -1.000000	1.000000, -1.000000, -1.000000
5	v 1.000000 -1.000000 1.000000	1.000000, -1.000000, 1.000000
6	v -1.000000 -1.000000 1.000000	-1.000000, -1.000000, 1.000000
7	v -1.000000 -1.000000 -1.000000	-1.000000, -1.000000, -1.000000
8	v 1.000000 1.000000 -0.999999	1.000000, 1.000000, -0.999999
9	v 0.999999 1.000000 1.000001	0.999999, 1.000000, 1.000001
10	v -1.000000 1.000000 1.000000	-1.000000, 1.000000, 1.000000
11	v -1.000000 1.000000 -1.000000	-1.000000, 1.000000, -1.000000
12	vn 0.0000 -1.0000 0.0000	


```

3 o icospnere
4 v 0.000000 -1.000000 0.000000 0.000000, -1.000000, 0.000000
5 v 0.723607 -0.447220 0.525725 0.723607, -0.447220, 0.525725
6 v -0.276388 -0.447220 0.850649 -0.276388, -0.447220, 0.850649
7 v -0.894426 -0.447216 0.000000 -0.894426, -0.447216, 0.000000
8 v -0.276388 -0.447220 -0.850649 -0.276388, -0.447220, -0.850649
9 v 0.723607 -0.447220 -0.525725 0.723607, -0.447220, -0.525725
10 v 0.276388 0.447220 0.850649 -0.276388, 0.447220, 0.850649
11 v -0.723607 0.447220 0.525725 -0.723607, 0.447220, 0.525725
12 v -0.723607 0.447220 -0.525725 0.276388, 0.447220, -0.850649
13 v 0.276388 0.447220 -0.850649 0.894426, 0.447216, 0.000000
14 v 0.894426 0.447216 0.000000 0.894426, -0.447216, 0.000000
15 v 0.000000 1.000000 0.000000 0.000000, 1.000000, 0.000000
16 v -0.162456 -0.850654 0.499995 -0.162456, -0.850654, 0.499995
17 v 0.425323 -0.850654 0.309011 0.425323, -0.850654, 0.309011
18 v 0.262869 -0.525738 0.809012 0.262869, -0.525738, 0.809012
19 v 0.850648 -0.525736 0.000000 0.850648, -0.525736, 0.000000
20 v 0.425323 -0.850654 -0.309011 0.425323, -0.850654, -0.309011
21 v -0.525730 -0.850652 0.000000 -0.525730, -0.850652, 0.000000
22 v -0.688189 -0.525736 0.499997 -0.688189, -0.525736, 0.499997
23 v -0.162456 -0.850654 -0.499995 -0.162456, -0.850654, -0.499995
24 v -0.688189 -0.525736 -0.499997 -0.688189, -0.525736, -0.499997
25 v 0.262869 -0.525738 -0.809012 0.262869, -0.525738, -0.809012
26 v 0.951058 0.000000 0.309013 0.951058, 0.000000, 0.309013
27 v 0.951058 0.000000 -0.309013 0.951058, 0.000000, -0.309013
28 v 0.000000 0.000000 1.000000 0.000000, 0.000000, 1.000000
29 v 0.587786 0.000000 0.809017 0.587786, 0.000000, 0.809017
30 v -0.951058 0.000000 0.309013 -0.951058, 0.000000, 0.309013
31 v -0.587786 0.000000 0.809017 -0.587786, 0.000000, 0.809017
32 v -0.587786 0.000000 -0.809017 -0.587786, 0.000000, -0.809017
33 v -0.951058 0.000000 -0.309013 -0.951058, 0.000000, -0.309013
34 v 0.587786 0.000000 -0.809017 0.587786, 0.000000, -0.809017
35 v 0.000000 0.000000 -1.000000 0.000000, 0.000000, -1.000000
36 v 0.688189 0.525736 0.499997 0.688189, 0.525736, 0.499997
37 v -0.262869 0.525738 0.809012 -0.262869, 0.525738, 0.809012
38 v -0.850648 0.525736 0.000000 -0.850648, 0.525736, 0.000000
39 v -0.262869 0.525738 -0.809012 -0.262869, 0.525738, -0.809012
40 v 0.688189 0.525736 -0.499997 0.688189, 0.525736, -0.499997
41 v 0.162456 0.850654 0.499995 0.162456, 0.850654, 0.499995
42 v 0.525730 0.850652 0.000000 0.525730, 0.850652, 0.000000
43 v -0.425323 0.850654 0.309011 -0.425323, 0.850654, 0.309011
44 v -0.425323 0.850654 -0.309011 -0.425323, 0.850654, -0.309011
45 v 0.162456 0.850654 -0.499995 0.162456, 0.850654, -0.499995
46 vn 0.1024 -0.9435 0.3151

```

```
11 v -1.000000 1.000000 -1.000000
12 vn 0.0000 -1.0000 0.0000
13 vn 0.0000 1.0000 0.0000
14 vn 1.0000 -0.0000 0.0000
15 vn 0.0000 -0.0000 1.0000
16 vn -1.0000 -0.0000 -0.0000
17 vn 0.0000 0.0000 -1.0000
18 s off
```



```

45 v 0.162456 0.850654 -0.499995
46 vn 0.1024 -0.9435 0.3151
47 vn 0.7002 -0.6617 0.2680
48 vn -0.2680 -0.9435 0.1947
49 vn -0.2680 -0.9435 -0.1947
50 vn 0.1024 -0.9435 -0.3151
51 vn 0.9050 -0.3304 0.2680
52 vn 0.0247 -0.3304 0.9435
53 vn -0.8897 -0.3304 0.3151
54 vn -0.5746 -0.3304 -0.7488
55 vn 0.5346 -0.3304 -0.7779
56 vn 0.8026 -0.1256 0.5831
57 vn -0.3066 -0.1256 0.9435
58 vn -0.9921 -0.1256 0.0000
59 vn -0.3066 -0.1256 -0.9435
60 vn 0.8026 -0.1256 -0.5831
61 vn 0.4089 0.6617 0.6284
62 vn -0.4713 0.6617 0.5831
63 vn -0.7002 0.6617 -0.2680
64 vn 0.0385 0.6617 -0.7488
65 vn 0.7240 0.6617 -0.1947
66 vn 0.2680 0.9435 -0.1947
67 vn 0.4911 0.7947 -0.3568
68 vn 0.4089 0.6617 -0.6284
69 vn -0.1024 0.9435 -0.3151
70 vn -0.1876 0.7947 -0.5773
71 vn -0.4713 0.6617 -0.5831
72 vn -0.3313 0.9435 0.0000
73 vn -0.6071 0.7947 0.0000
74 vn -0.7002 0.6617 0.2680
75 vn -0.1024 0.9435 0.3151
76 vn -0.1876 0.7947 0.5773
77 vn 0.0385 0.6617 0.7488
78 vn 0.2680 0.9435 0.1947
79 vn 0.4911 0.7947 0.3568
80 vn 0.7240 0.6617 0.1947
81 vn 0.8897 0.3304 -0.3151
82 vn 0.7947 0.1876 -0.5773
83 vn 0.5746 0.3304 -0.7488
84 vn -0.0247 0.3304 -0.9435
85 vn -0.3035 0.1876 -0.9342
86 vn -0.5346 0.3304 -0.7779
87 vn -0.9050 0.3304 -0.2680
88 vn -0.9822 0.1876 0.0000
89 vn -0.9050 0.3304 0.2680
90 vn -0.5346 0.3304 0.7779
91 vn -0.3035 0.1876 0.9342
92 vn -0.0247 0.3304 0.9435
93 vn -0.5746 0.3304 0.7488

```

C:\Program Files (x86)\Python37-32\python.exe

```

0.102400, -0.943500, 0.315100
0.700200, -0.661700, 0.268000
-0.268000, -0.943500, 0.194700
-0.268000, -0.943500, -0.194700
0.102400, -0.943500, -0.315100
0.905000, -0.330400, 0.268000
0.024700, -0.330400, 0.943500
-0.889700, -0.330400, 0.315100
-0.574600, -0.330400, -0.748800
0.534600, -0.330400, -0.777900
0.802600, -0.125600, 0.583100
-0.306600, -0.125600, 0.943500
-0.992100, -0.125600, 0.000000
-0.306600, -0.125600, -0.943500
0.802600, -0.125600, -0.583100
0.408900, 0.661700, 0.628400
-0.471300, 0.661700, 0.583100
-0.700200, 0.661700, -0.268000
0.038500, 0.661700, -0.748800
0.724000, 0.661700, -0.194700
0.268000, 0.943500, -0.194700
0.491100, 0.794700, -0.356800
0.408900, 0.661700, -0.628400
-0.102400, 0.943500, -0.315100
-0.187600, 0.794700, -0.577300
-0.471300, 0.661700, -0.583100
-0.331300, 0.943500, 0.000000
-0.607100, 0.794700, 0.000000
-0.700200, 0.661700, 0.268000
-0.102400, 0.943500, 0.315100
-0.187600, 0.794700, 0.577300
0.038500, 0.661700, 0.748800
0.268000, 0.943500, 0.194700
0.491100, 0.794700, 0.356800
0.724000, 0.661700, 0.194700
0.889700, 0.330400, -0.315100
0.794700, 0.187600, -0.577300
0.574600, 0.330400, -0.748800
-0.024700, 0.330400, -0.943500
-0.303500, 0.187600, -0.934200
-0.534600, 0.330400, -0.777900
-0.905000, 0.330400, -0.268000
-0.982200, 0.187600, 0.000000
-0.905000, 0.330400, 0.268000
-0.534600, 0.330400, 0.777900
-0.303500, 0.187600, 0.934200
-0.024700, 0.330400, 0.943500
0.574600, 0.330400, 0.748800
0.794700, 0.187600, 0.577300
0.889700, 0.330400, 0.315100
0.306600, 0.125600, -0.943500
0.303500, -0.187600, -0.934200
0.024700, -0.330400, -0.943500
-0.802600, 0.125600, -0.583100
-0.794700, -0.187600, -0.577300
-0.889700, -0.330400, -0.315100
-0.802600, 0.125600, 0.583100
-0.794700, -0.187600, 0.577300
-0.574600, -0.330400, 0.748800
0.306600, 0.125600, 0.943500
0.303500, -0.187600, 0.934200
0.534600, -0.330400, 0.777900
0.992100, 0.125600, 0.000000

```



```
Left subset: 6  
Right subset: 6  
Left subset: 4  
Right subset: 2  
Left subset: 2  
Right subset: 2  
Left subset: 2  
Right subset: 4  
Left subset: 3  
Right subset: 1
```

C:\Program Files

Left subset: 40
Right subset: 40

Left subset: 19
Right subset: 21

Left subset: 8
Right subset: 11

Left subset: 4
Right subset: 4

Left subset: 2
Right subset: 2

Left subset: 2
Right subset: 2

Left subset: 6
Right subset: 5

Left subset: 4
Right subset: 2

Left subset: 2
Right subset: 2

Left subset: 2
Right subset: 3

Left subset: 10
Right subset: 11

Left subset: 5
Right subset: 5

Left subset: 3
Right subset: 2

Left subset: 2
Right subset: 3

Left subset: 6
Right subset: 5

Left subset: 2
Right subset: 4

Left subset: 2
Right subset: 2

Left subset: 3
Right subset: 2

```
C:\Program Files (x86)\Python37-32\python.exe
Camera Position: 0.000000, 0.000000, -10.000000
Number of differing origins: 0
```

15

```
C:\Program Files (x86)\Python37-32\python.exe
Camera Position: 1.000000, 4.000000, 42.000000
Number of differing origins: 0
```

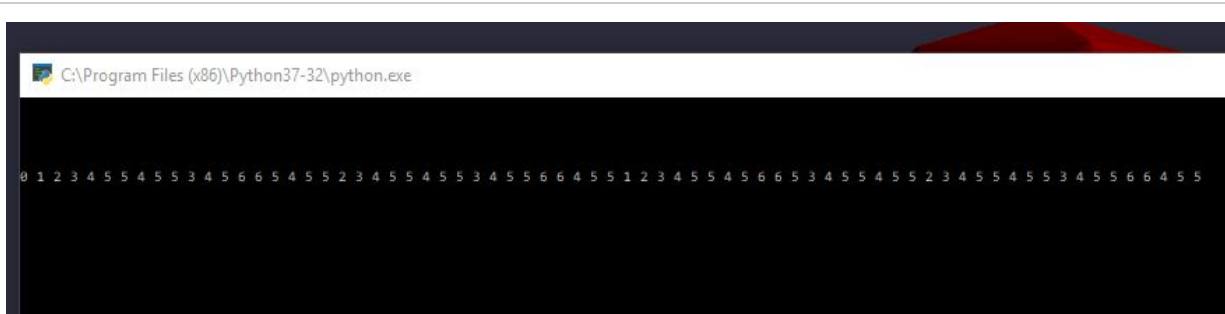
16

```
C:\Program Files (x86)\Python37-32\python.exe
Camera Position: 0.000000, 0.000000, 0.000000
Number of differing origins: 0
```

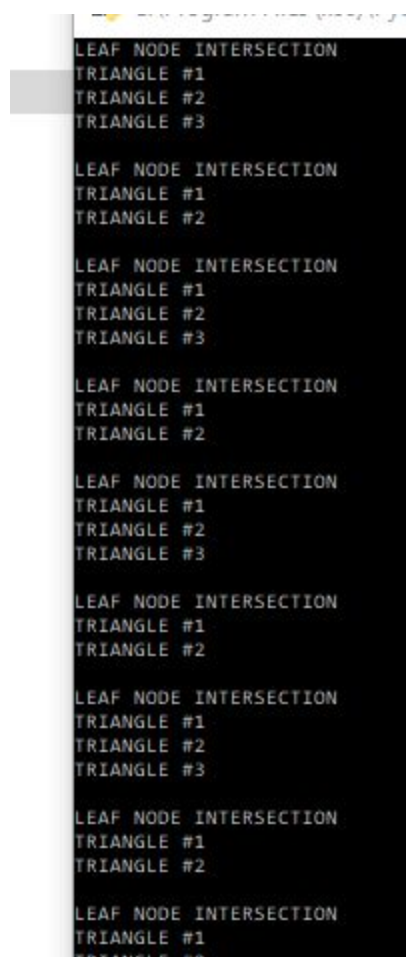
17

```
0 1 2 3 3 2 1 2 2 3 3
```

18



19



20

C:\Program Files (x86)\Python3\

```
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

LEAF NODE INTERSECTION
TRIANGLE #1
TRIANGLE #2

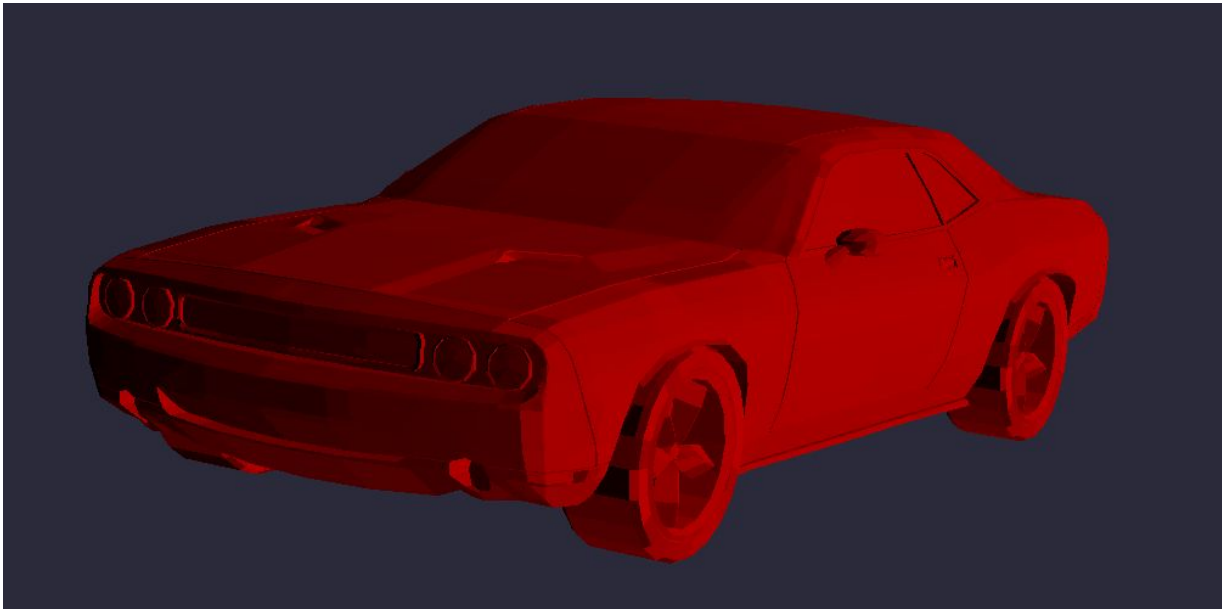
LEAF NODE INTERSECTION
TRIANGLE #1
```

9.087555
9.087929
9.088306
9.088687
9.089068
9.089453
9.089839
9.090226
9.090617
9.091009
9.091404
9.091801
9.092199
9.092601
9.093004
9.093410
9.093817
9.094226
9.094638
9.095052
9.095469
9.095887
9.096308
9.096730
9.097154

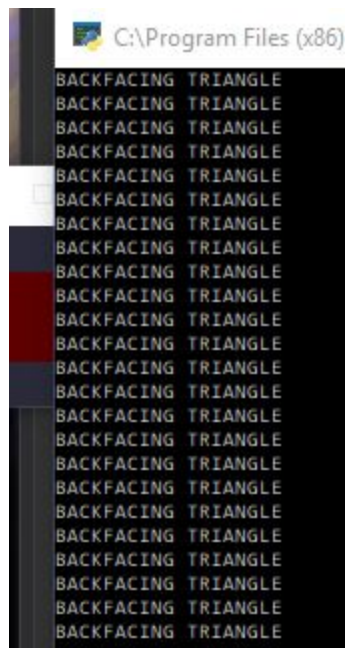
22

9.401798
9.484606
9.710418
9.531639
9.423002
9.349417
9.277775
9.208036
9.173598
9.142825
9.139098
9.136259
9.134309
9.133244
9.148196
9.195458
9.273114
9.352924
9.434942
9.579527
9.950927
9.811505
9.504904
9.376772
9.303849
9.232849
9.184466
9.151492
9.119614
9.103765

23



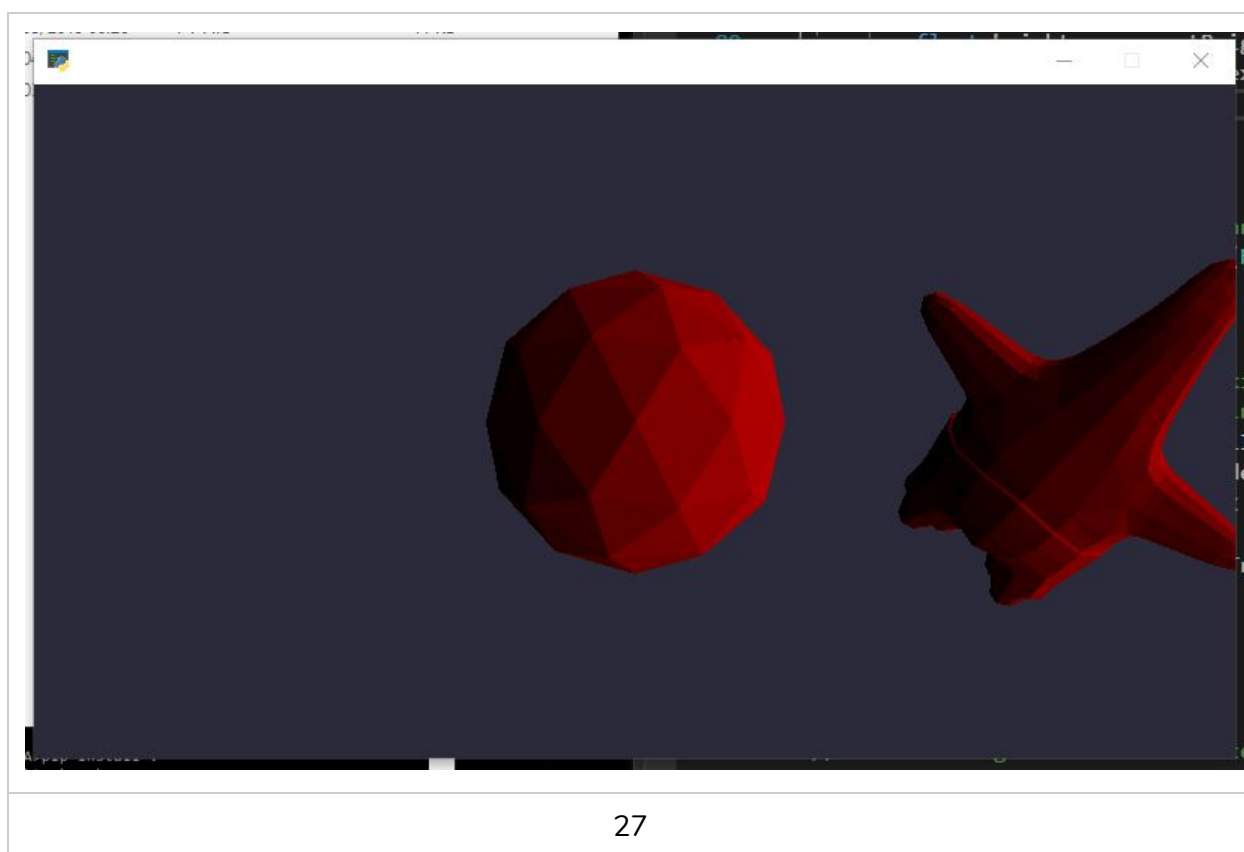
24



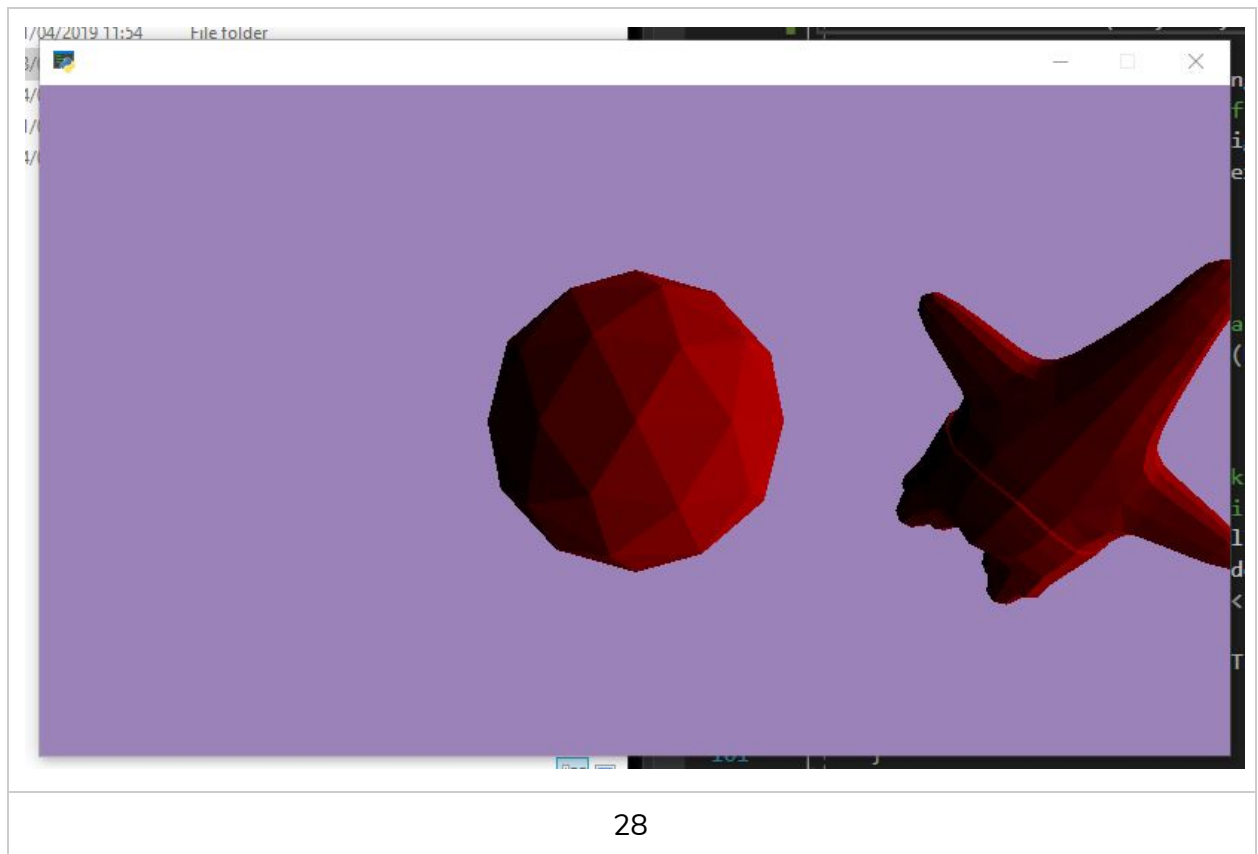
25

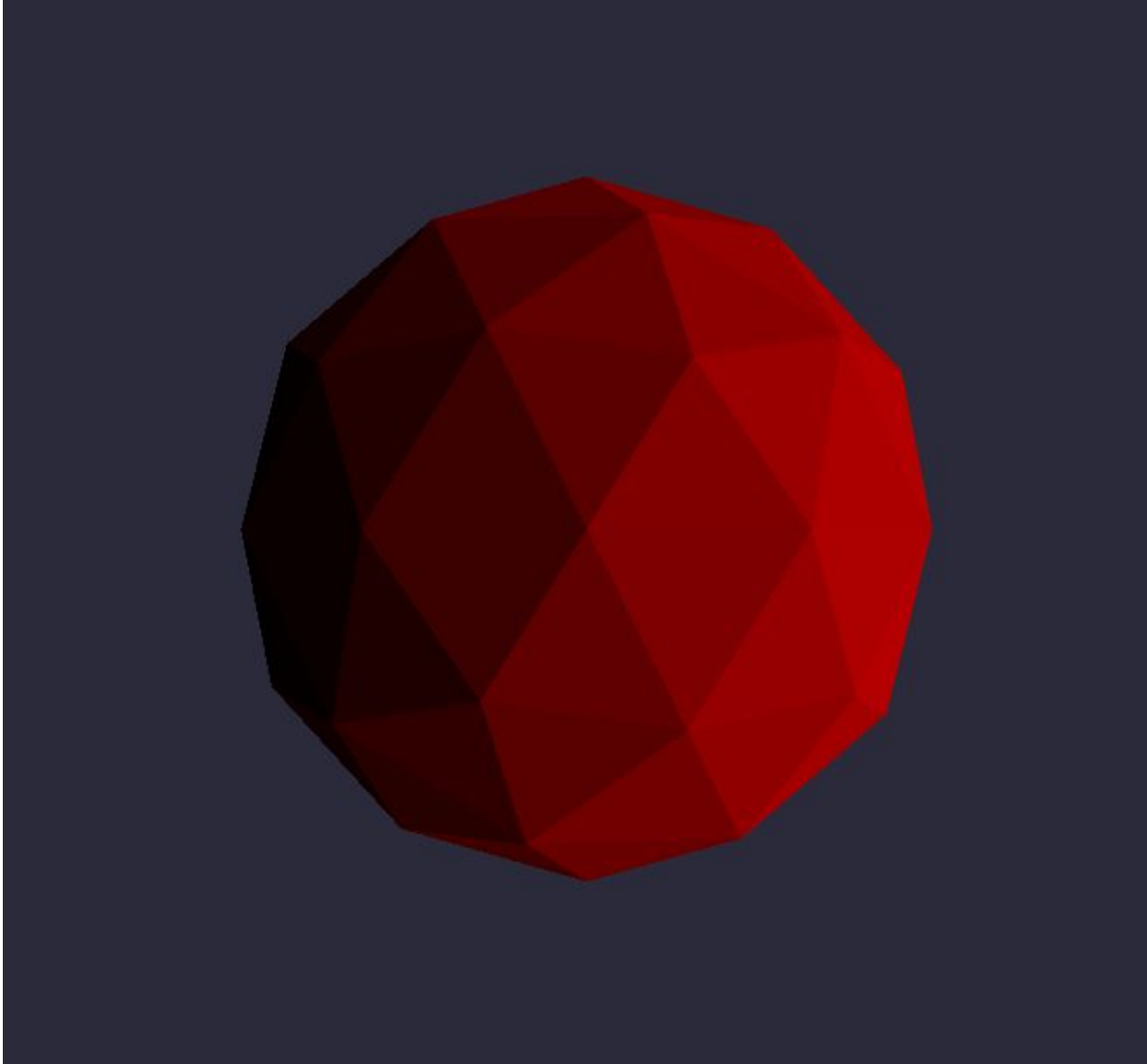


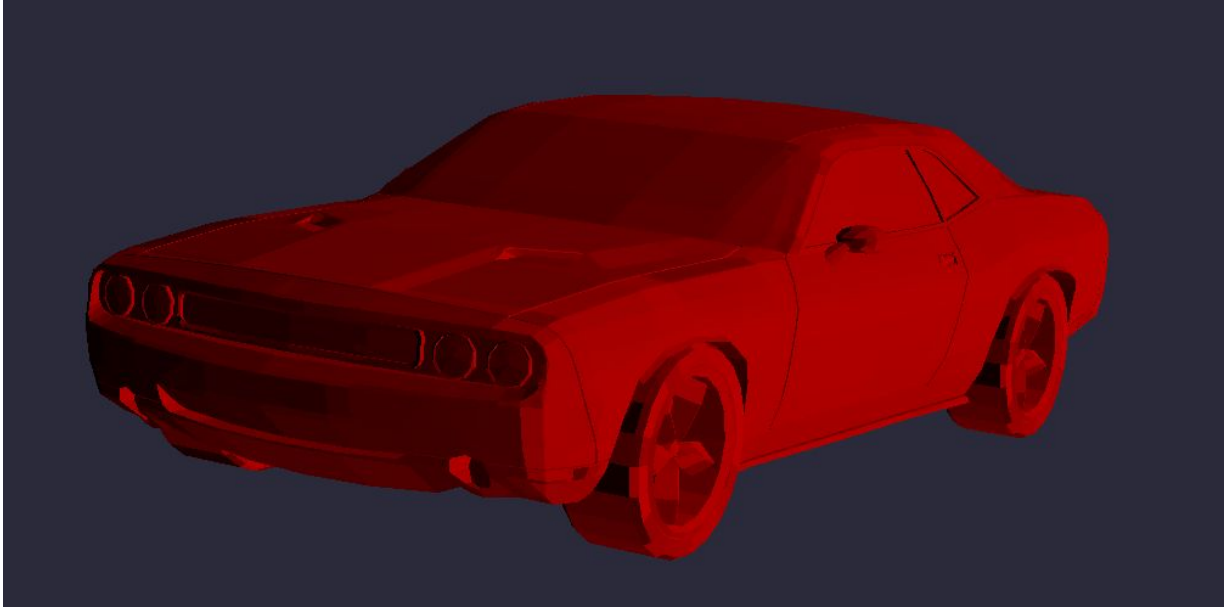
26



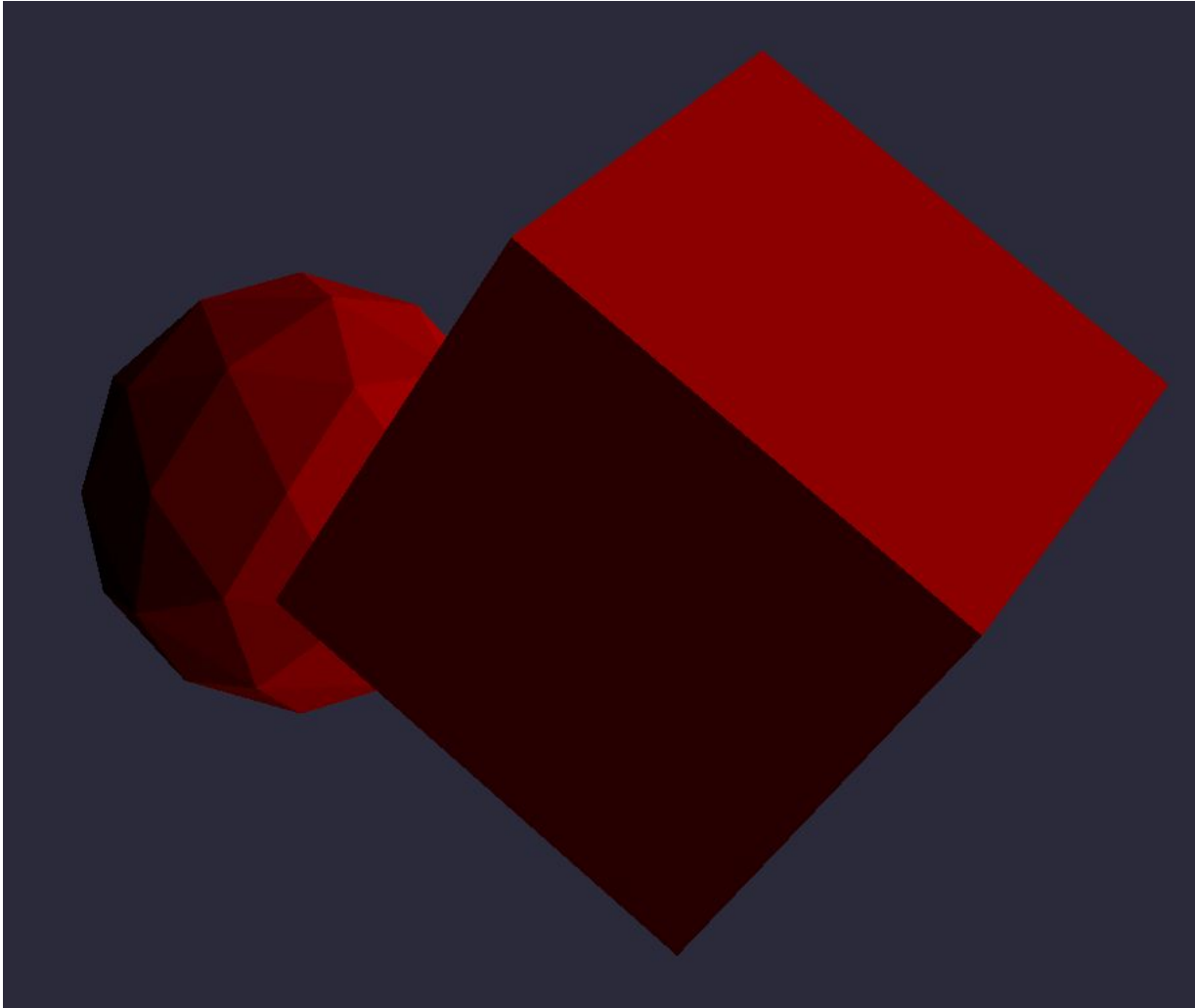
27

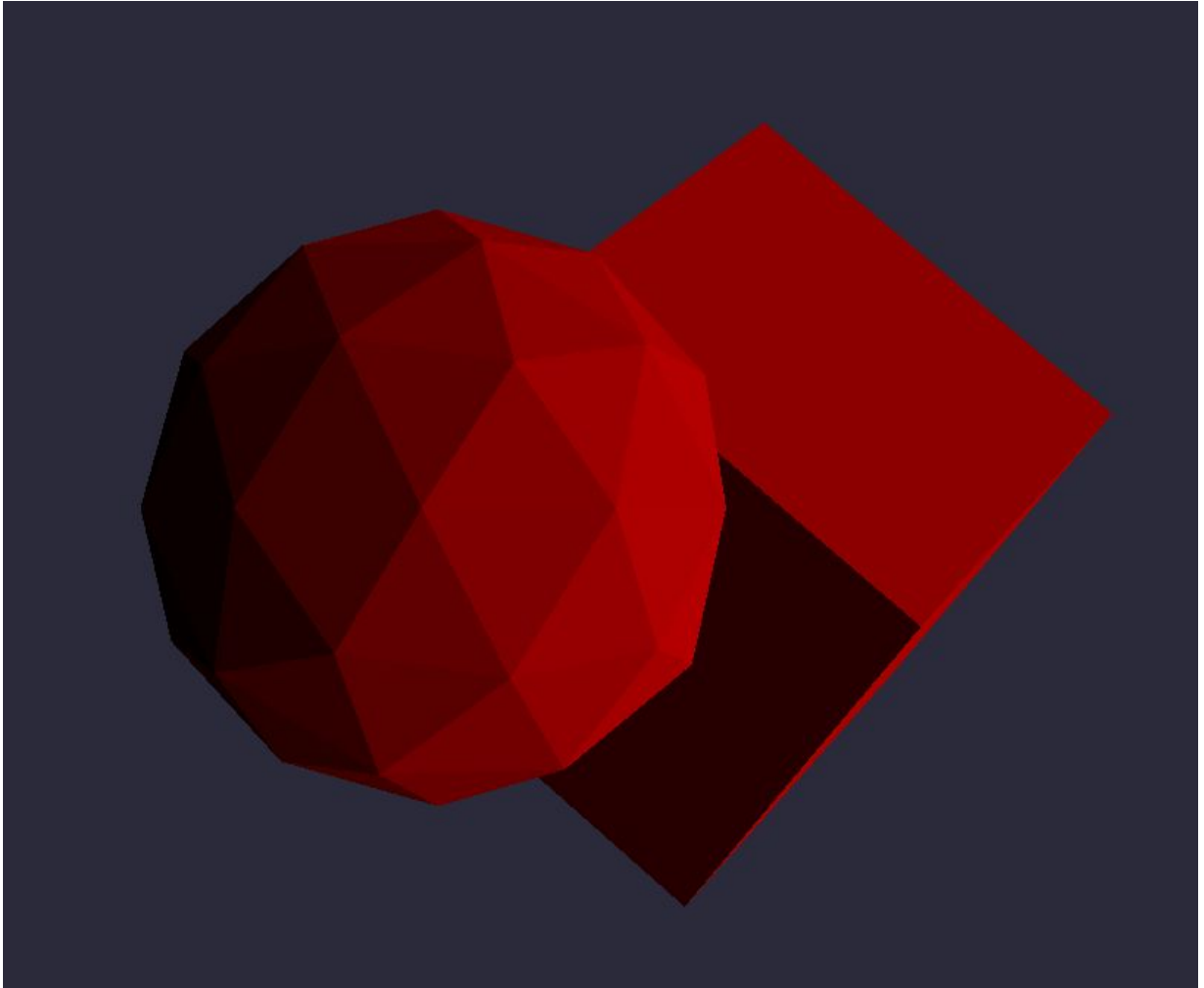


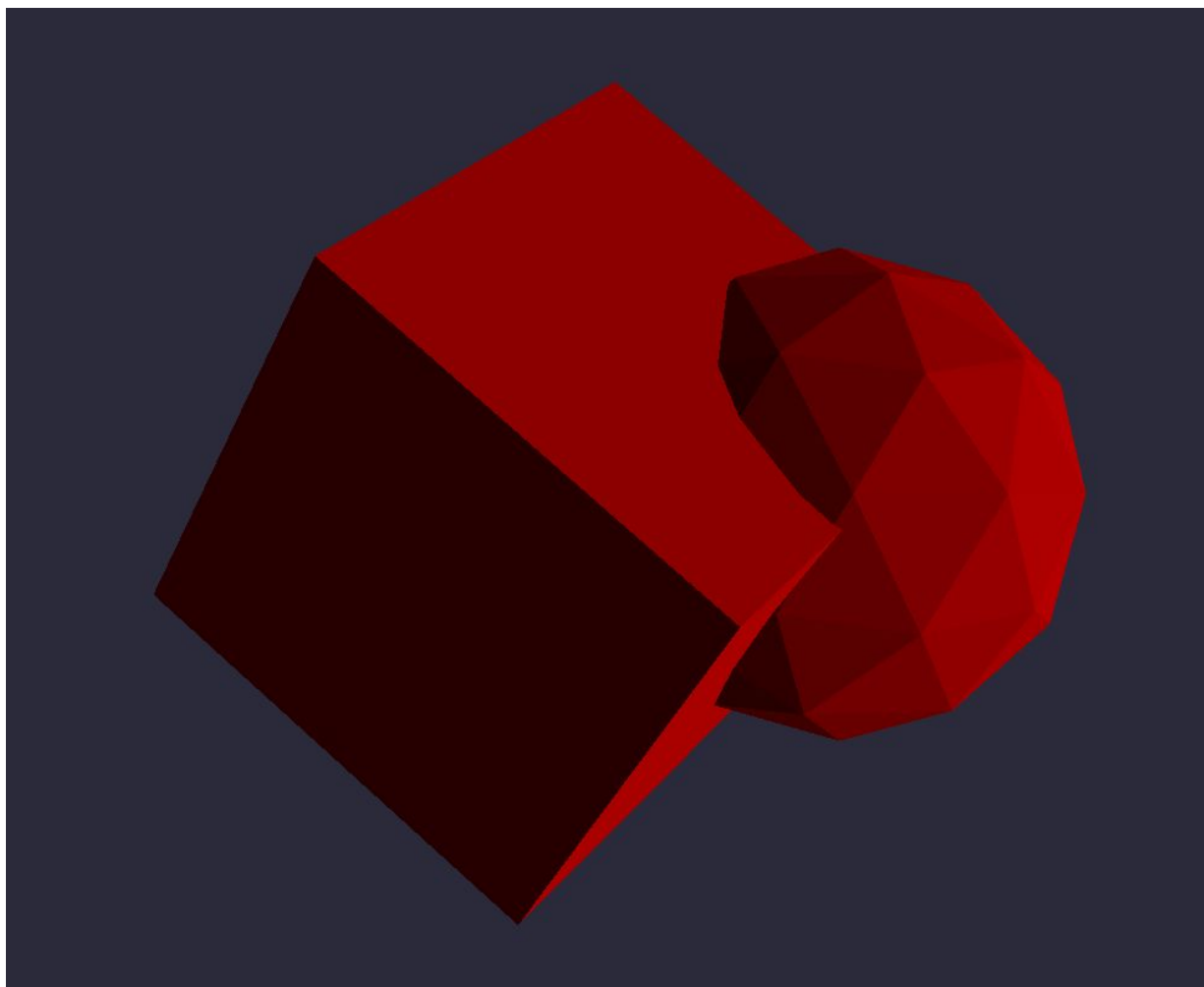




30

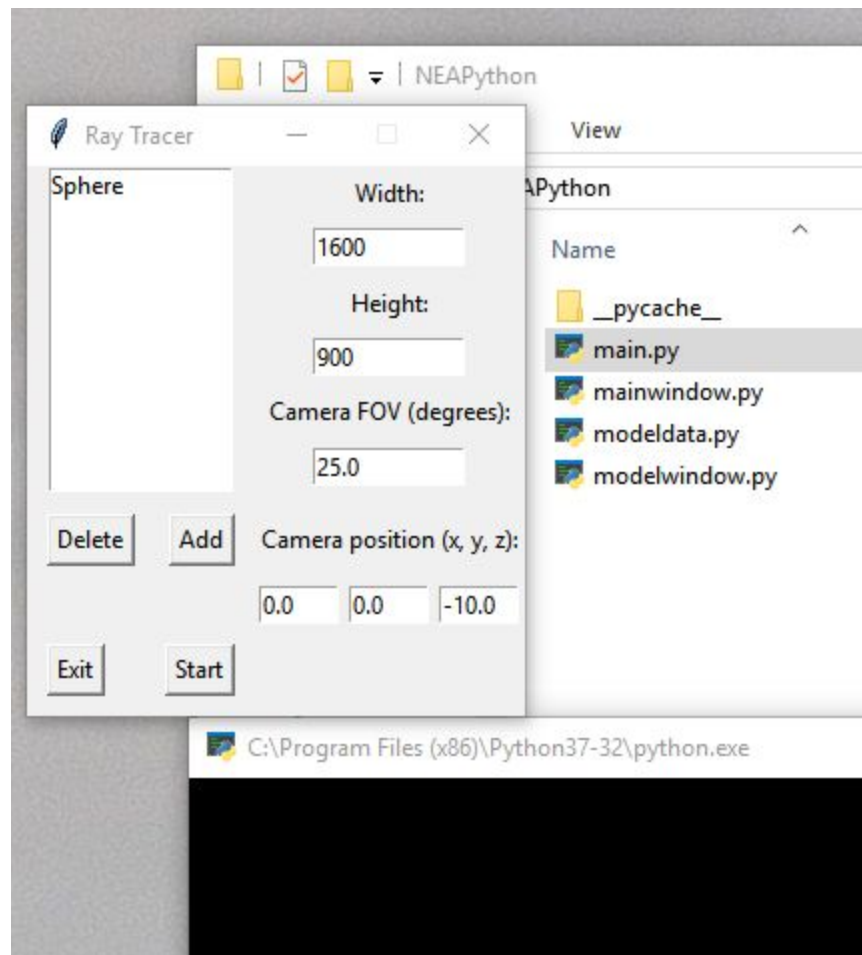


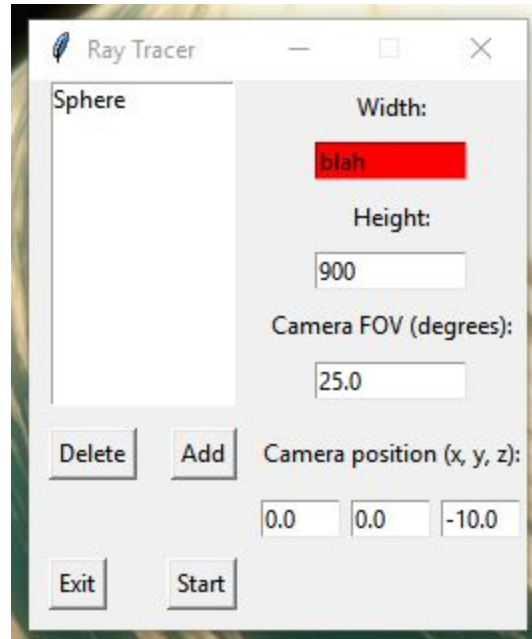




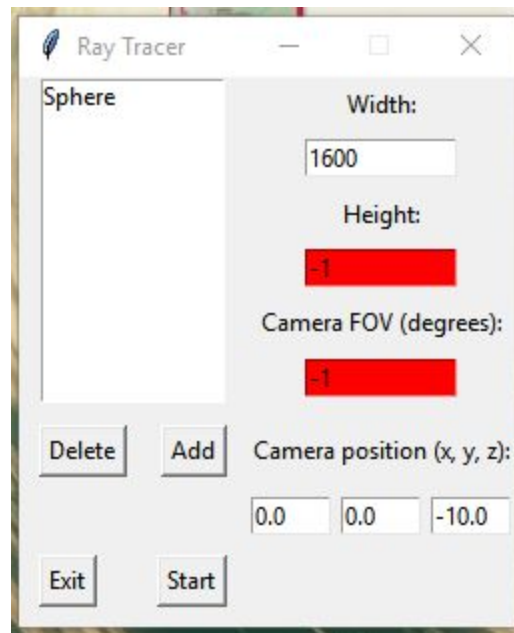
33

GUI

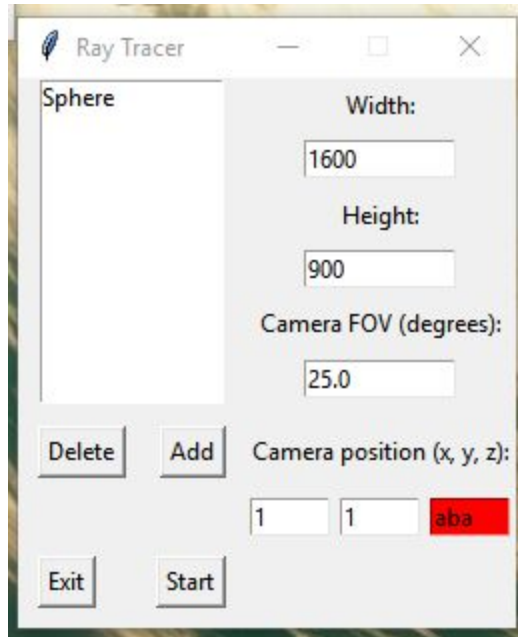




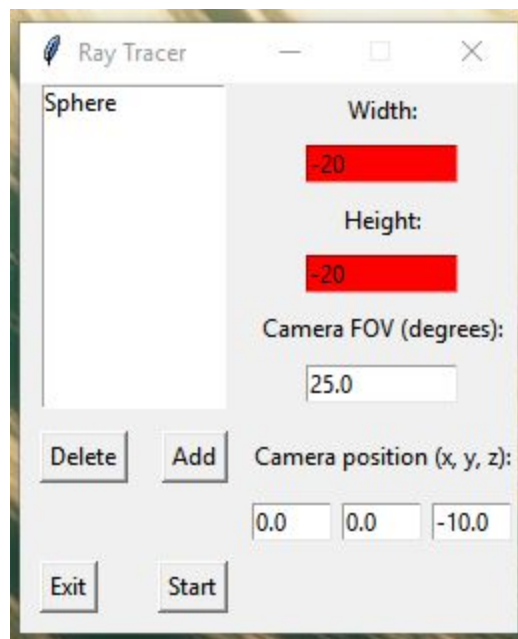
2



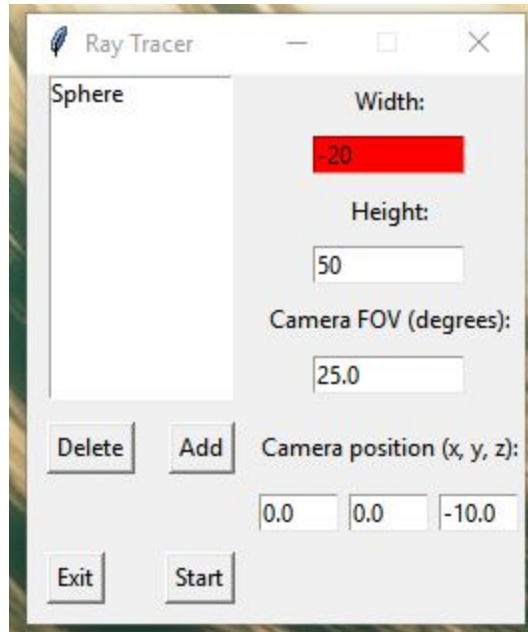
3



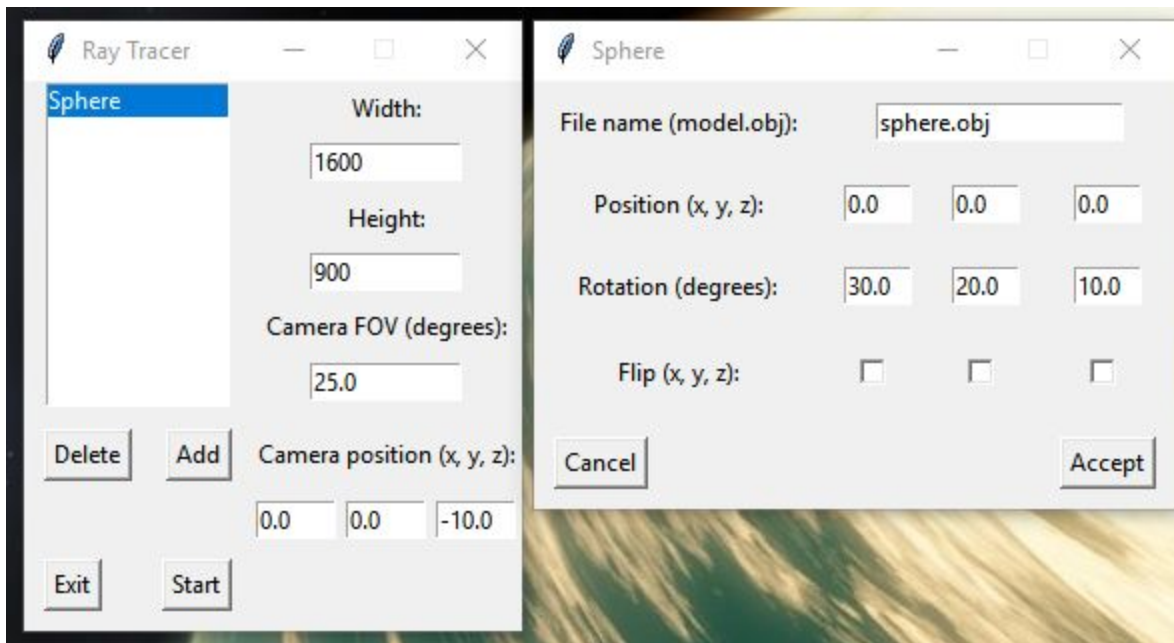
4



5



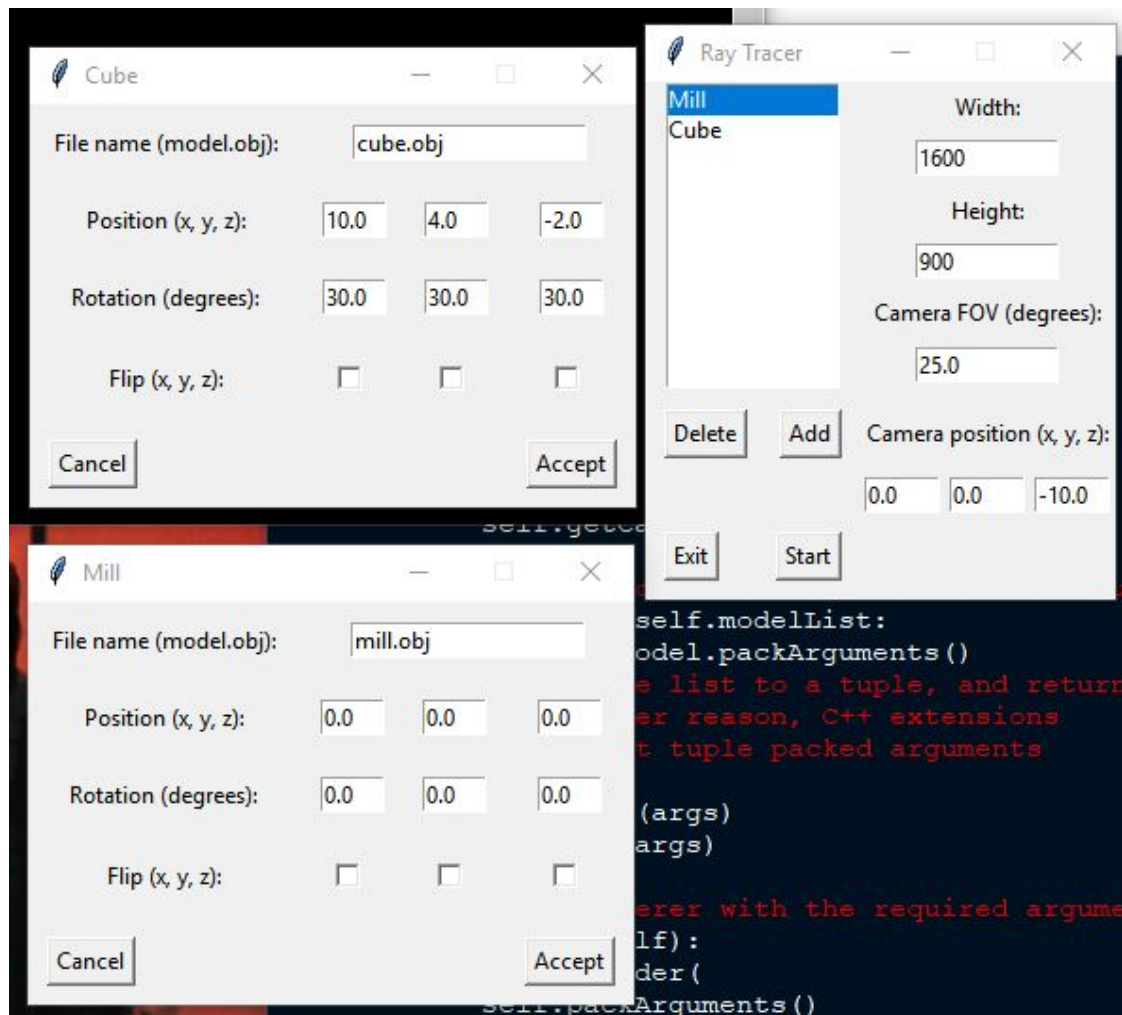
6



7

```
[1600,
900,
0.0,
0.0,
-10.0,
25.0,
'sphere.obj',
0.0,
0.0,
0.0,
30.0,
20.0,
10.0,
False,
False,
False]
```

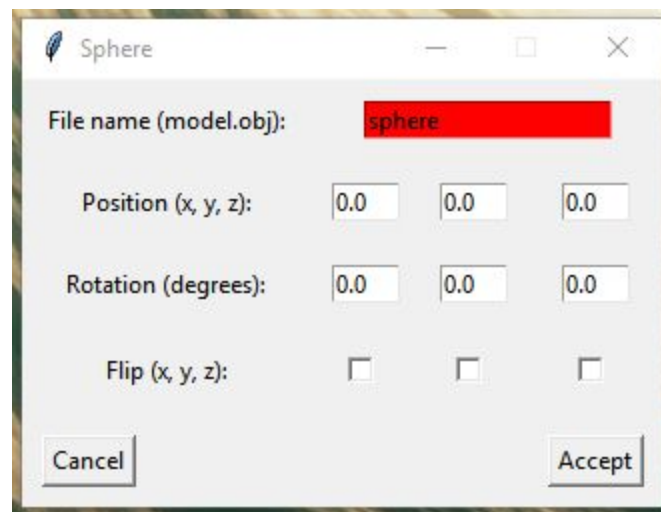
8



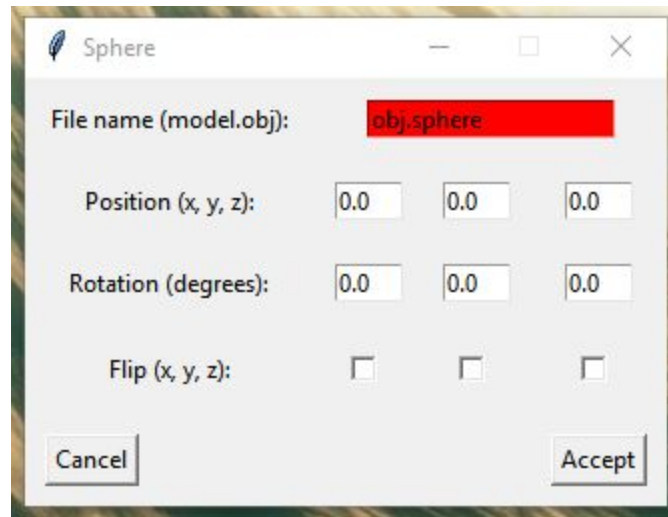
9

```
[1600,  
900,  
0.0,  
0.0,  
-10.0,  
25.0,  
'mill.obj',  
0.0,  
0.0,  
0.0,  
0.0,  
0.0,  
0.0,  
False,  
False,  
False,  
'cube.obj',  
10.0,  
4.0,  
-2.0,  
30.0,  
30.0,  
30.0,  
False,  
False,  
False]
```

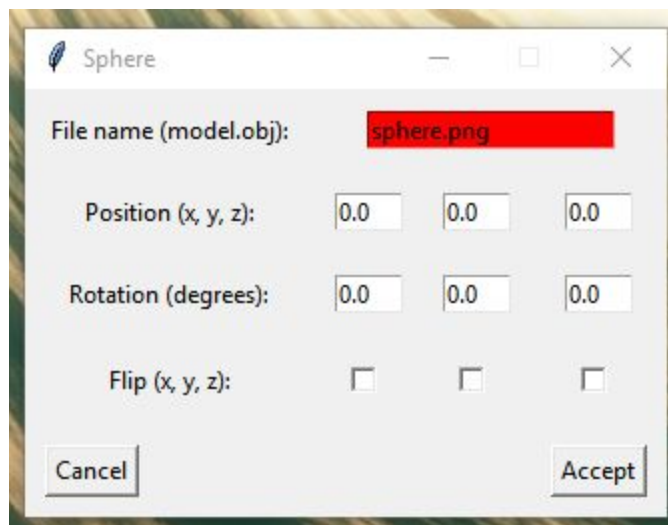
10




11



12



13

 Sphere


File name (model.obj):

Position (x, y, z):

Rotation (degrees):

Flip (x, y, z): ☐ ☐ ☐

14

 Sphere

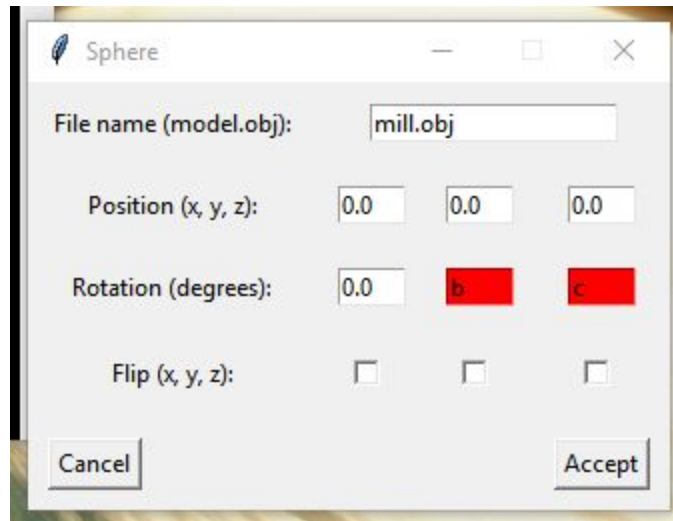
File name (model.obj):

Position (x, y, z):

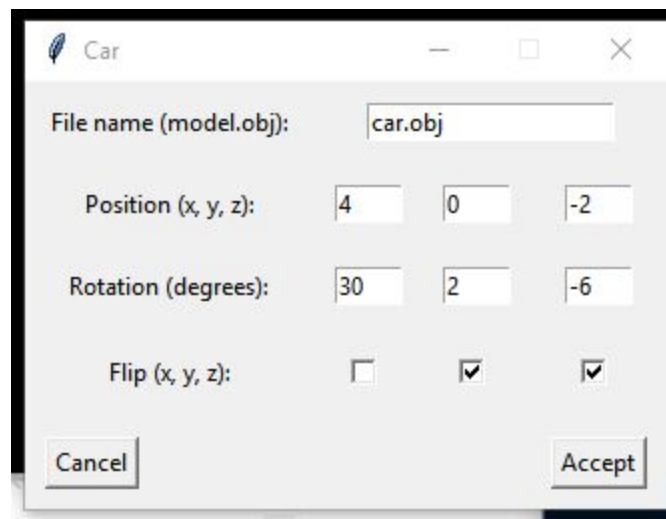
Rotation (degrees):

Flip (x, y, z): ☐ ☐ ☐

15



16



17

```
C:\Program Files (x86)\Python\32\python.exe
['car.obj', 4.0, 0.0, -2.0, 30.0, 2.0, -6.0, False, True, True]
```

18

Evaluation

Fulfillment of requirements

“Provide a graphical user interface”

The GUI satisfies all of the requirements, including “customisation of variables”, and being able to include “a variable number of models to be added, and their positions and orientations in world space to be specified”.

The renderer can be started from the GUI, and the user-defined variables are sent to the C++ application. Inputted data is checked for validation, and if invalid, the renderer is prevented from starting.

“Parse triangle meshes from external ‘OBJ’ files”

The OBJ file parser satisfies all the requirements, vertex, normal, and index data are read from the file into lists, in the correct order. Indices are put into ‘Triangle’ objects.

“Store the triangle meshes in BVHs, one per model”

The storage of models satisfies all the requirements, as vertex and normal data are stored by the model itself, and each model has a BVH, which it passes the list of ‘Triangle’ objects to.

“Recursively spatially partition triangles”

The spatial partitioning system satisfies all the requirements, as the set of triangles is recursively spatially partitioned into spatially distinct, evenly sized subsets, which improves the efficiency of ray intersections with the hierarchy.

“Render models using BVHs”

The renderer satisfies all of the requirements. Rays are emitted from the camera origin through the projection plane to create a rectilinear projection, minimising the ‘fish-eye’ effect. These rays test for collisions with bounding volumes recursively, and when a leaf node is intersected with, the triangles themselves are tested against.

When multiple triangles overlap, the closest triangle is used. Triangle back faces are not rendered, and if no triangles are intersected with, the background colour is used.

“Directional shading”

The shading satisfies all of the requirements, as triangles facing the light source are brighter than those facing away, and there is a smooth transition from light to dark.

“Render multiple models at once”

Multiple models can be rendered in the same scene, so this requirement is satisfied.

Overall conclusion

Overall, all of the requirements stated at the beginning of the project were included in the project to a satisfactory standard.

However, if I was to start the project again, I would include a few more requirements:

- Allow models to be resized in the model window, to change their size in world space and thus the rendered image. Models exported from blender can have unexpected sizes and my project doesn't account for this.
 - This would be achieved by extending the 'Transform' class with scaling coefficients in each axis. A new function would be written to create a 3x3 matrix with the scaling coefficients along the diagonal, and all other elements as zero. This is the scaling matrix, and could be absorbed into the resultant matrix of the 'transform' instance easily.
- Allow the direction of the camera to be specified, currently only the position of the camera can be changed, allowing the direction or even the orientation to be changed would give flexibility to the user.
 - This could be achieved by changing the current 'position offset' technique for camera movement currently used into a matrix transformation similar to that used by the 'transform' class. This means that vertices will be read in in model-space, transformed to world-space by the model-specific 'transform', and then transformed to the camera's view space by the camera transform. While this would require some effort to extend in this way, no structural changes to the existing program will have to be made.

- Change the flat list representation of models in the scene into a scene-level BVH for efficiency. While this has little to no effect for small number of models, as the number of models increases it becomes more and more efficient over the flat list.
 - This should be relatively easy to do, as a model-level BVH already exists and can likely be retrofitted for scene-level use with little modification to the overall structure.
- The rendering process could be multithreaded to improve speed, as it is a largely parallel process.
 - This could be achieved by using the C++ 'windows.h' and 'process.h' header files for the threading functionality, or even utilising the GPU by using CUDA. CPU bound multithreading and CUDA especially would require massive rewrites of the code, and changes to the fundamental structure of the program to gain any benefit. It would be very difficult to do.
- Texture mapping could be implemented, using imported textures with the included UV coordinates in OBJ files.
 - This could be achieved fairly easily, as long as a reliable library for reading texture files is found. Blender exports UV coordinates by default, so it would be relatively easy to have a texture lookup, especially as the barycentric coordinates of intersections are found as a byproduct of the intersection algorithm.
- Shading could be changed from a flat shading model to a Gourad shading model to increase the smoothness of meshes that are attempting smoothness, for example a sphere. To go further, the Phong model could also be implemented, including a specular component which helps create definition in the model.
 - To achieve this, the normal vectors of vertices would have to be interpolated across triangles, instead of using a single face normal as is currently implemented. This would require some reworking of the triangle and BVH data structures, as well as the lighting algorithm.

Examples of final renders

