Comparing Rasterisation to Raytracing for the Future of Real-Time 3D Rendering

- Benjamin Schaaf

Abstract

The question of whether Raytracing will replace Rasterization in the future of real-time rendering is investigated. Rasterisation and raytracing are explained and compared. It was established that raytracing is theoretically superior to rasterization both in terms of realism and features, as well as scaling.

Both methods were implemented using Python, OpenGL and OpenCL to run on current graphics hardware, an NVIDIA GeForce GTX 470 chipset, and benchmarked with common 3D models.

It was found that rasterization showed no performance impact for all cases that were still testable with raytracing. It was finally concluded that although superior, current graphics hardware is not able to run raytracing due to the way branching is handled. Once the hardware is able to cope with raytracing, it would only be a matter of time before raytracing becomes the standard for 3D real-time rendering.

Contents

1. Introduction

Three dimensional (3D) rendering is a highly researched and very complex area of computer graphics. It has many applications, such as: simulations, manufacturing, video games and movies. There have been many developments of different 3D rendering methods that range from being very fast and unrealistic, to very realistic and slow, each of which have different applications because of these properties. Only one of these methods has become the de-facto standard of modern day, real-time 3D rendering: Rasterisation.

Through constant hardware improvements, a vast amount of processing power is now available, in contrast to when rasterisation became omnipresent. This might give us opportunities to use more powerful 3D rendering methods in the same kinds of applications rasterisation excels in. When looking at the list of the most common 3D rendering methods, the next most obvious candidate, in terms of speed, is raytracing.

When looking at the future, we always need to consider how technology might evolve and how we could use this new technology, in order to guide our development. 3D graphics is a very important field, which has been ever growing and prevalent in our lives. It is therefore important to consider what the future of graphics looks like.

In this essay, both of these rendering methods will be explored, explaining their key differences and discussing which method might prevail over the future of real-time 3D rendering within the context of standard graphics acceleration hardware.

2. Rasterisation

Similar to vector graphics, rasterisation algorithms work by performing certain transformations on the 3D data to create a 2D representation. It attempts to emulate the way a camera would work through a simplified mathematical model. The algorithm transforms the data from 3D world space into 2D screen space, thus making the 3D rendering problem a simple vector graphics problem. The 3D data, usually represented as triangles, is transformed to 2D and then into individual pixels. In order to circumvent the problem of overlapping triangles in the 2D space, the 'depth' of each pixel is also calculated, remembered, and compared before writing new pixels.

The process used in current hardware can be described as follows:

1. Every vertex of every triangle is transformed from object space to clip space. Object space is described by a coordinate system that is relative to the objects center. Clip space is described by a coordinate system that is relative to the camera's view space, described by a near-clip and far-clip plane, as well as a view-of-view angle and the camera's own transformations. Clip space is restricted to a [-1, 1] boundary on all axis. The transformation from object space to world space is described by a transformation matrix, while the transformation between world space

and clip space is described by a projection matrix. Clip space is used to clip vertices that are outside of the boundaries, as an optimisation.

2. Every vertex, now in clip space, is translated to normalised device coordinates (NDC). These consist of 2D values ranging from 0 to 1, where [0, 0] is the bottom left of the 'device', while [1, 1] is the top right. The device is synonymous to the render target, some sort of raster[1]. Although the normalised device coordinates are 2D, they are expressed using 3D vectors, such that the third value represents the logarithmic distance from the virtual camera. The reason the distance is logarithmic is in order to prioritise depth accuracy for close up rendering.

3. After this, all the triangles are rasterised, using methods such as scanline rendering[2], and converted into individual fragments. For every fragment a color is evaluated, through a process called shading. This can include lighting calculations, specular highlighting and the like.

Rasterisation of a single triangle in 3D space:

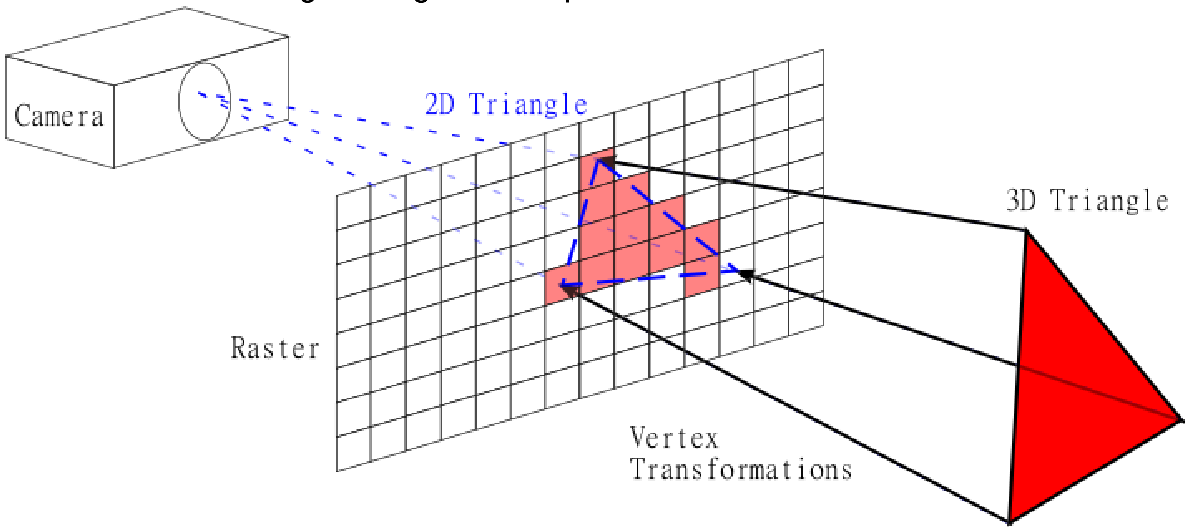


Image created using Inkscape[29] and Gimp[30]

The entire process can be highly parallelised, since the transformation of each triangle vertex and the shading of each pixel are completely independent operations. This nature gave birth to the invention of the GPU, the Graphics Processing Unit, which is optimised to perform the same actions many times with small variances in input, ie. transforming the vertex of each triangle from 3D space to 2D space.

Along with the GPU and graphics cards came the creation of two competing standards for communicating and performing actions on them: D3D[6] (Direct3D) and OpenGL[5] (Open Graphics Library), by Microsoft and The Khronos Group respectively. Both of these standards were originally completely fixed function based, but along with the evolution of graphics cards both standards have given more and more customisation to the rendering process to the point where even the generation of the 3D data can be performed on the GPU.

1 A 2D grid of pixels.
2 A method of converting vector graphics to raster graphics.

3. Raytracing

Raytracing can be compared to the old greek theories on how we see light. In 300 BC Euclid, a greek mathematician, hypothesised that light is the result of a beam travelling from the eye. Just like in Euclid's theories, raytracing works by performing intersection tests between rays extending out from the ‘eye’ or the camera and all objects in the environment. From the intersection point, the color at that point can be determined and then written to the target of the rendering. This is done for every pixel on the screen.

Formally, the process can be described as follows:

1. For every pixel to be rendered, generate a mathematical ray, ie. a point and direction in world space. This can be done using projection transformations of a virtual camera. This can be visualised as having a point camera with rays traveling through a raster in-front of the camera.

2. Find the intersection of the ray to all scene geometry that is closest to the origin of the ray. Without any optimisations, and assuming we only have triangle data, this means performing a ray-triangle intersection test for all triangles in the scene.

3. Given the intersection point, evaluate the color at that point using the environment, and then write that color into the target pixel. The color is usually evaluated using similar techniques to rasterisation, part of which is shading. This step however also includes evaluation of reflections, shadows, refractions and transparency.

Raytracing of a single triangle in 3d space:

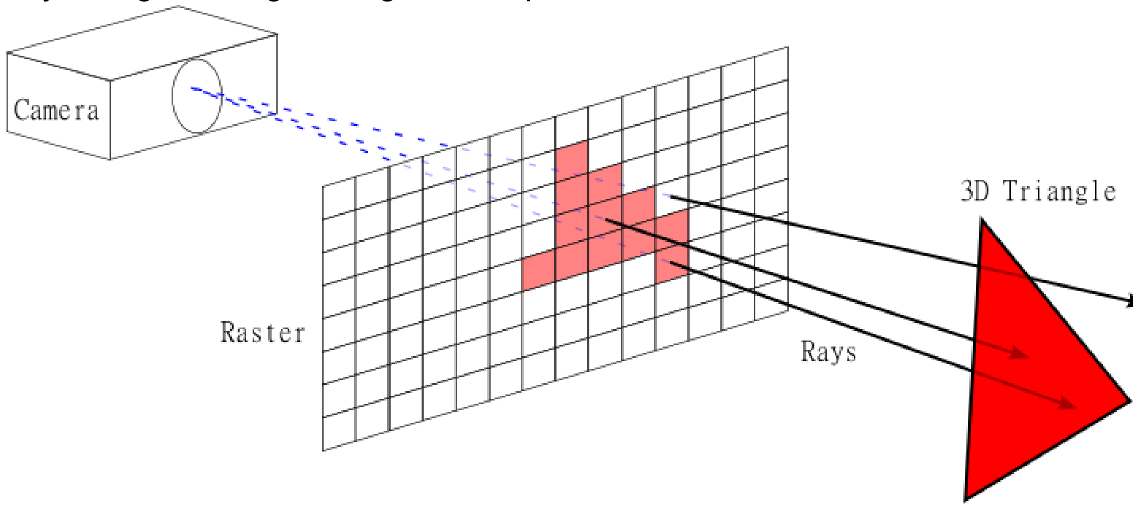


Image created using Inkscape[29] and Gimp[30]

Similarly to rasterisation, raytracing can also be easily parallelised, since the rendering of each individual pixel is not dependent on that of any other. However, unlike rasterisation, raytracing has had little dedicated hardware or standards and is therefore, usually performed on the CPU.

Ideally the technology used for rasterisation could also be used for raytracing. Raytracing would seemingly benefit just as much from using a GPU as rasterisation. Thankfully graphics cards have moved entirely beyond the rendering world and into the area of generalised computing. With the relatively recent standards, OpenCL[7] (Open Computing Language) and CUDA[8] (Compute Unified Device Architecture), developed by The Khronos Group and NVIDIA respectively, programmers are now able to use modern GPUs for many kind of non-graphical parallel tasks.

4. Why Raytracing?

Raytracing has a number of inherent advantages over rasterisation that allow it to not only follow reality more closely, mimicking (more or less) the behaviour of light, but raytracing also opens the door to a more powerful rendering system, allowing for effects not feasible with rasterisation, such as transparent shadows and non-planar reflections.
Most of the advantages of raytracing come from the fact that raytracing uses imaginary light rays instead of vector based transformation. Raytracing is therefore closer to a simulation of real light, as opposed to the approximation offered by rasterisation, making it easier to replicate common behaviours of light.

4.1 Transparency

With rasterisation, transparency is always an issue. By only keeping the depth of each single pixel of the output, rasterisation is restricted to being able to draw one surface for every pixel. Transparency is therefore an issue because it relies on drawing one or more surfaces to a single pixel by combining rather than overriding. To circumvent this, transparent triangles are usually drawn after all opaque ones and are manually sorted before being rendered. This still causes issues and makes it impossible to render intersecting, transparent surfaces properly. Raytracing inherently does not have this problem because it follows a ray of light. When a ray's intersection is calculated and such intersection is with a transparent object, it is trivial to continue the intersection test with objects after the first intersection and evaluate the total resulting color.

4.2 Shadows

In order to create the effects of shadows in normal rasterisation one would usually render out a shadow map, which would involve re-rendering the entire scene again, possibly more than once. With raytracing, shadows are relatively inexpensive, as they simply involve performing another intersection test from the hit point towards the light source. With slight adjustments this can be further improved to also allow transparent surfaces to cast shadows, something infeasible using rasterisation due to the numerous problems of transparency with rasterisation.

4.3 Reflection

Perfect reflection, ie. physically accurate, in rasterisation is feasible as long as the reflection is caused by a plane, as the scene can simply be rendered again using a different projection, coming from the plane. Reflection involving non-planar surfaces can also be achieved by rendering a spherical reflection from a single point, typically the centre of an object, and then using that to look up the reflection for the entire object. This can lead to nice looking, but none the less highly inaccurate reflections.

For raytracing, in a similar way to shadows, reflection can be achieved by performing another intersection test out from the hit point along the reflection vector. The resulting color can then be mixed in with the already evaluated color, resulting in relatively cheap reflections. This also means that reflection with raytracing works for any sort of surface, including spheres.

4.4 Refraction[3]

Similarly to reflection, rasterisation can only achieve planar refraction using a similar concept, but again, refraction is relatively cheap, and more accurate, with raytracing. Not only is it done with yet another intersection test, but since most refractions happen through a solid, not just a surface, a more realistic refraction could be achieved by performing another intersection test for the exit point from the solid.

4.5 Anti-Aliasing[4]

One thing rasterisation excels at is anti-aliasing. Because of the transformation from triangle data to pixel data, it is relatively simple to perform standard anti-aliasing techniques, such as multisampling and super-sampling, giving some very nice results. With raytracing, anti-aliasing is a simple, but very expensive process. Since no edge-data can be obtained from an intersection, anti-aliasing usually involves using more than one ray for every pixel, usually a multiple of 4. This means that, with anti-aliasing, a single render could take anywhere from 4, to 16+ times[5] longer, than without anti-aliasing. In effect, this is the same as rendering a higher resolution image and downsizing.

4.6 Depth of Field[6]

A Depth of Field effect with rasterisation is a relatively inexpensive task, usually applying a certain amount of blur, depending on the depth of each pixel, for every pixel, after the scene has been rendered. The physically accurate equivalent for raytracing would, similarly to anti-aliasing, also involve using more than one ray for every pixel. By using the same method as rasterisation, and also writing the depth of each pixel, the same effect can be achieved in a less accurate, but less expensive way.

4.7 Other Shapes

In general, rasterisation is restricted to shapes that can be transformed from 3D space to 2D space without losing shape, however more specifically, both OpenGL and D3D restrict the 3D data to planar polygons, which in many cases are triangulated before rendering.

[3] The change in direction of light caused by it traveling between substances with different densities.
[4] Reducing the distortion caused by representing vector graphics using pixels. In effect, making edges look smoother.
[5] A common amount of multisampled anti-aliasing.
[6] An effect causing objects at distances further away from a focal point to become more blurred. This is caused by the way eyes and camera's work, but has to be emulated with rasterisation and raytracing.

Raytracing is much less restricted. Since the only action that interacts with the 3D data is an intersection test with a ray, the data is simply restricted to geometry where ray-intersection tests are possible. This includes anything from standard triangle data, to a sphere, to something like a 3D Julia fractal:

Raytraced render of a 3D Julia fractal[37]

4.8 Efficiency

Rasterisation can bottleneck in many parts of the process. This can be anything from the fill-rate, ie. how many pixels the graphics card can actually draw per second, or the sheer amount of 3D data that needs to be rendered. This means it is difficult to determine exactly everything that influences how long a render will take. However with various optimisations, such as hidden surface determination, where it is first determined whether an object can be seen before it is rendered, we can safely say that rendering a scene will be in terms of the number of triangles:

$$O(\mathit{number\ of\ triangles}) = O(n)$$

Raytracing on the other hand is quite the opposite. With the most basic algorithm, one obvious term is the actual number of pixels that need to be rendered:

$$O(\mathit{number\ of\ pixels}) = O(n)$$

So raytracing in effect must be in terms of the width and height of the rendering target. However, this is a simplistic view, the most expensive part of any raytracing algorithm is the intersection test. Unoptimised, this test also runs in terms of the number of triangles N, which results in a worse time than rasterisation:

$$O(\mathit{number\ of\ pixels} * \mathit{number\ of\ triangles}) = O(mn)$$

However, the intersection tests could be optimised by eliminating intersection tests with triangles that are impossible to get hit by each ray. Such an optimisation is accomplished through the use of spatial lookup trees, which subdivide triangles into separate spaces. Triangles can then be eliminated by doing a single intersection test with one of their parents in the tree.

An example of such a structure is an Octree, which subdivides space into 8 equal leafs, each containing triangles and other subdivisions. Using an Octree, the worst case time is still $O(n)$, but the average time is brought down to $O(1)$. This means that raytracing can actually run entirely in terms of the number of pixels:

$$O(number\ of\ pixels) = O(n)$$

Other than the memory of the actual input and output data, both rendering algorithms are relatively light-weight in terms of memory usage. Except for the optimisations mentioned above, the memory used by either algorithm is usually constant and very small, since most of it is temporary. All input data is directly converted to output, there is no historical tracking.

4.9 Implementation

When it comes to actually using the graphics card to perform rendering using rasterisation, the process is very simple. Both OpenGL and D3D are specifically designed to be used for rasterisation and although the latest versions allow more generalised computing, their sole purpose is still to be used for 3D rendering.

For raytracing however, as previously mentioned, no such standard APIs exist. The entire implementation has to be created from scratch. OpenCL and CUDA only give the minimum functionality needed to compile and upload code to the GPU and then run it from the CPU, along with some functionality allowing memory to be transferred. Since OpenCL and CUDA are still relatively new technologies, and are generally less used than OpenGL or D3D, they come with their fair share of driver bugs and other implementation faults that also need to be dealt with.

5. Process

In order to compare how both rasterisation and raytracing perform while both running on the GPU, it was decided to implement both using standard approaches. The language chosen for both implementations was python[1], which provides bindings for all of the aforementioned APIs, enabling simpler implementation.

OpenGL was chosen as the rasterisation API over DirectX due to it being more portable, while OpenCL was chosen over CUDA, because CUDA is only supported by NVIDIA graphics cards, while OpenCL is much more widely supported. PyOpenGL[3], PyOpenCL[4] and pygame[2] libraries were used for the relevant API interaction.

Because of the large array of areas that could be covered in both rendering systems, the problem space for both systems was narrowed down to opaque, static scenes with only basic surface shading. To keep both comparable, the scene data was further restricted to solely triangle data.

From this, three variables could be derived. The first being the number of triangles that are being rendered, the second, the number of pixels the final image is made up of, and the final, being whether or not all triangles were in the camera's view or not.

Both implementations could take advantage of the reduced problem space and introduce optimisations not possible for all types of scenes. To make the comparison as fair as possible, as many of these optimisations as possible were introduced for both rendering systems.

5.1 The System

Scene data is loaded from a '.obj' file using the Wavefront OBJ format[7]. The file itself is generated using Blender[31].

The system uses a generic interface to communicate with both renderers. When the system starts, it first creates a window with an OpenGL context of a given size. It then requests the given renderer to perform setup operations given the size of the created context.

Once the program enters the main loop, it requests the renderer to perform a single render operation given certain camera conditions. The rendered frame is then drawn to the screen and the entire time is added to a total. The program runs for a certain number of frames and then exits, returning the average amount taken to render a frame.

5.2 The Rasteriser

Because graphics cards are already highly optimised for rasterisation, to the point where the actual rasterisation process is a single hardware function, there was no need to implement any high-level optimisation techniques. The rasteriser uses a standard forward rendering[28] as opposed to the more complicated deferred rendering[15][20][21] for simplicity.

To match the requirements of a static scene, all triangle data was uploaded individually as a single object. This, although more memory intensive, reduces the number of OpenGL calls needed for the rendering process. It also makes the GLSL shaders a lot simpler, as they do not have to use any object specific data.

5.3 The Raytracer

As graphics cards are not built to perform raytracing and are especially bad at branching, the implementation focuses on minimising the number of branches and calculations done for each pixel.

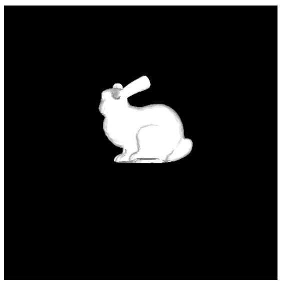
As an optimisation, the ray-intersection tests use an octree[18][26] structure to reduce the number of triangle intersection tests performed for every pixel.

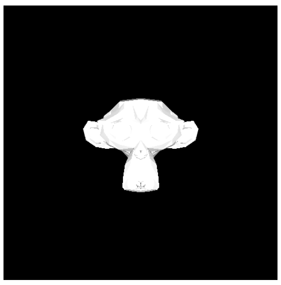
Because the scene would only be static, the octree could be generated in python in the setup phase and then passed to OpenCL for rendering.

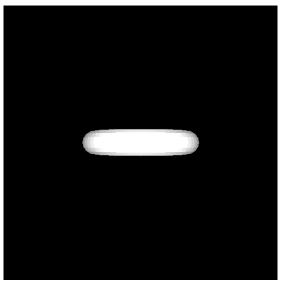
[7] An open, plain text file format for geometry definitions.

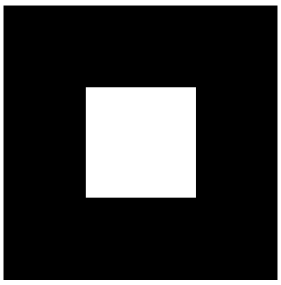
6. Results

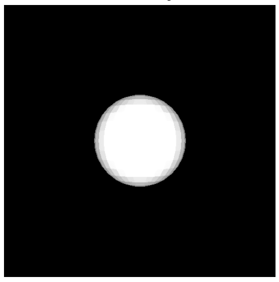
Tests:

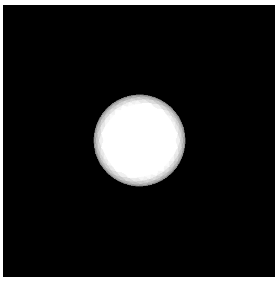

Stanford Bunny


Suzanne

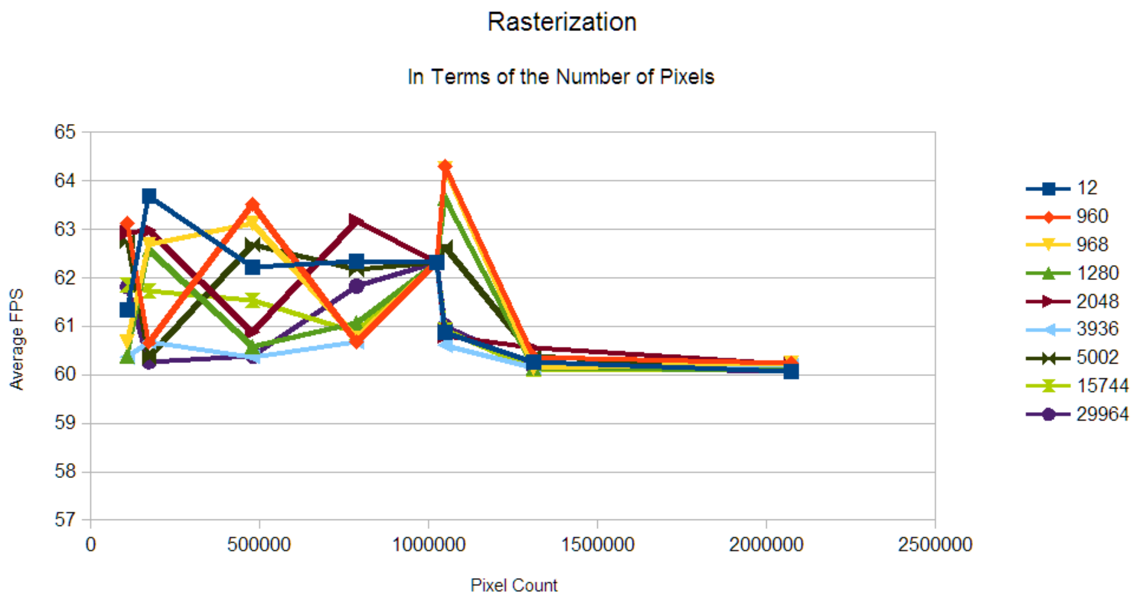

Torus


Unit Cube

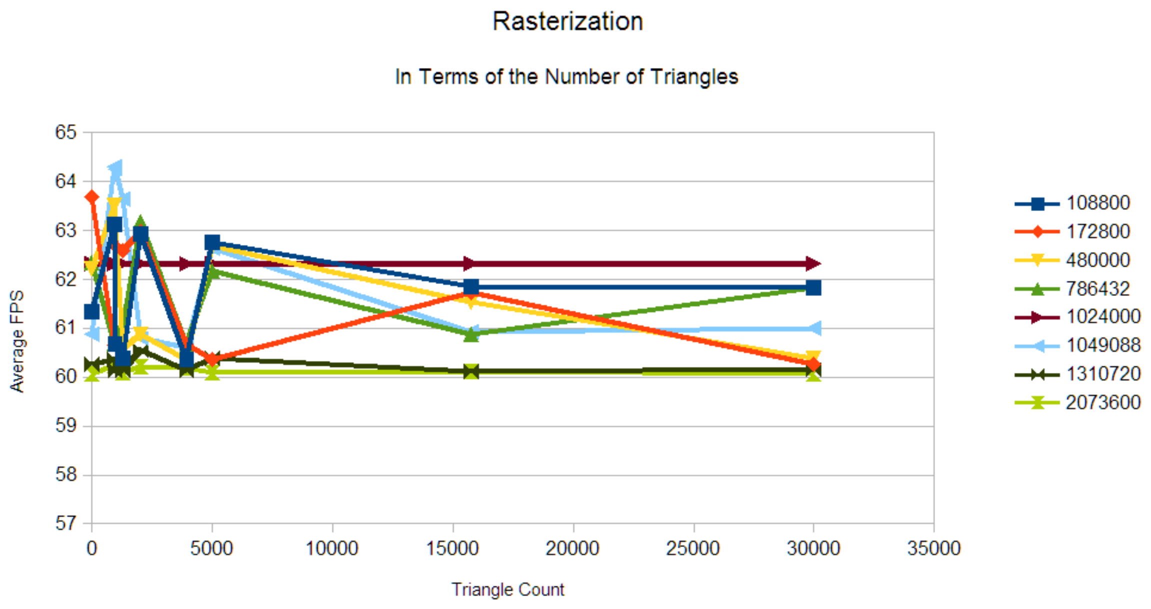

UV-Sphere


IcoSphere

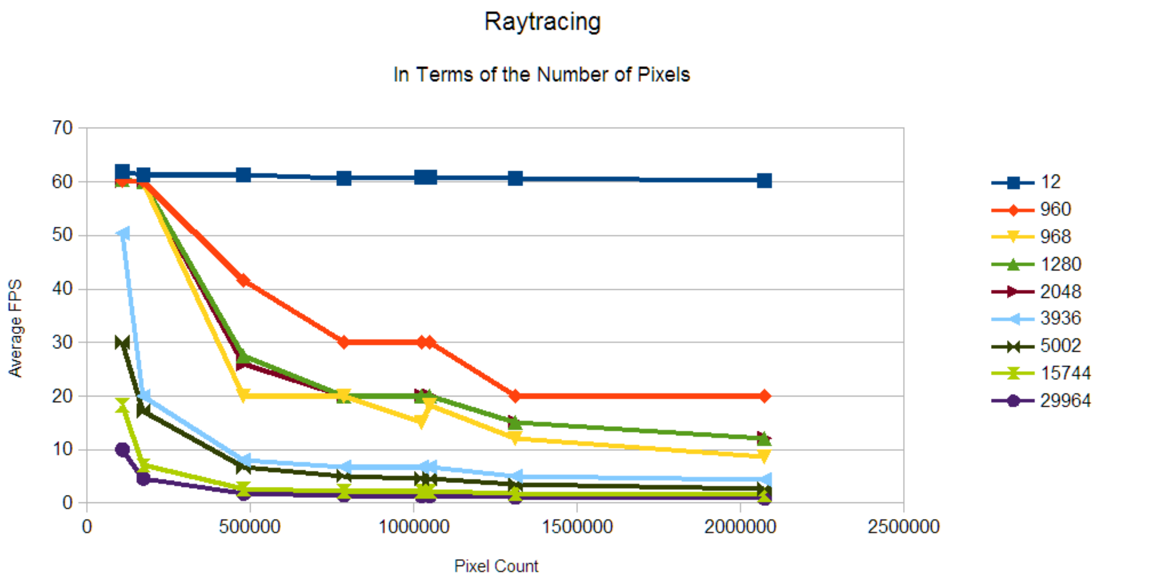
Benchmarks were taken using the above base models, with certain variations with a higher triangle count. They were then performed with varying standard screen sizes on a computer running on Windows 7 with an NVIDIA GeForce GTX 470 chipset. Each test was then run for 20 seconds and the average frame rate achieved was logged.



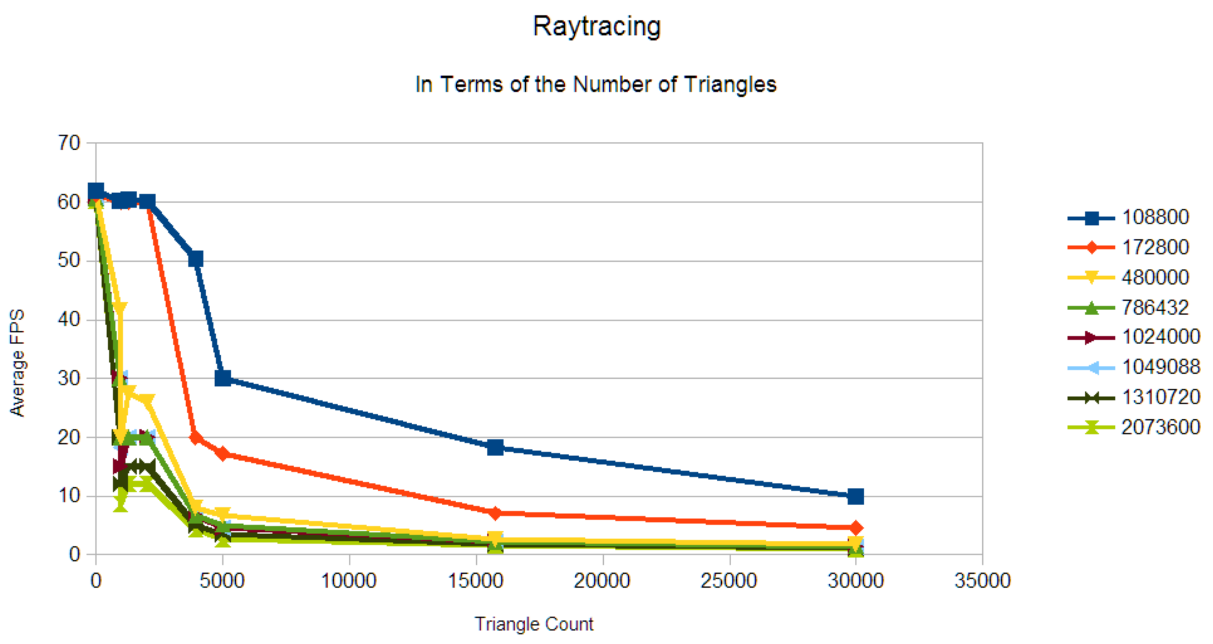
The frame rate that was achieved at different screen resolutions using rasterisation held consistently above 60 fps (frames per second), a limit seemingly imposed by pygame or OpenGL on the test system. This meant that for all test cases, there was no significant deviation from the maximum possible framerate.



When varying the number of triangles for rasterisation, there was again, no significant deviation from the maximum possible frame rate of 60fps. Although rasterisation theoretically scales badly in terms of the number of triangles, the test cases only included models up to approximately 30,000 triangles, a relatively small amount for rasterisation.



In contrast to rasterisation, raytracing shows some obvious trends. Although deviating from the theoretically expected linear trend, it does show how the number of pixels impacts rendering speed. It's also interesting to note that the red (960 triangles) and yellow (968 triangles) curves deviate greatly from each other, showing that the implementation has some other scaling issues that could be causing performance issues.



Even with the octree optimisation, raytracing showed obvious performance issues in terms of the number of triangles. Although theoretically $O(1)$, the octree lookup and intersection tests still had a significant performance impact when run on the graphics card.

7. Discussion

The results for rasterisation were exactly as expected. With all test cases, rasterisation performed at or above the maximum frame rate, 60fps, with no visible trends. With the simple shading that was used and considering that the graphics card used is generally able to render millions of triangles with very complicated, multi-pass shaders at the same frame rate, these results were completely predictable. None of the test cases were complicated enough to drive the hardware to its limits.

In contrast, raytracing had significant performance issues for all but the most simple test cases. This comparatively huge difference could be attributed to two likely root causes: Branching on the GPU and the optimisation that was used. In addition, graphics cards are not optimised to perform raytracing, which also leads to a general performance decrease.

A general problem with current graphics card hardware is the way branching is handled. Because the GPU are SIMD (Single Instruction, Multi-Data), when a branch is encountered all cores generally have to stop execution as long as they don't follow the same path. As raytracing and especially the octree implementation used rely heavily on branching and the implementation executes once per ray, meaning a large divergence in the results of branching, it is very likely that the performance issues experienced are purely due to the way the graphics card works. If the GPU were MIMD (Multi-Instruction, Multi-Data), then there would undoubtedly be better performance for raytracing.

The implementation of the octree is however also not as optimal as possible. In order to circumvent the lack of a stack on the GPU, the implementation uses a depth first search for the octree, testing all leafs that the ray intersects with in no particular order. If the intersection tests were ordered, such that leafs closer to the ray's origin were tested first, some intersection tests could have possibly been avoided.

Although the biggest current hinderance to raytracing is hardware development, it is not out of the question for future hardware to perform much better with the same tests, or even rival rasterisation. If the problems with branching were to be circumvented, raytracing would probably have much better results in comparison.

As per data gathered by w3school.com[9], in terms of the browser display statistics, it can be seen that over time the size of displays has been getting larger, but over the past years this growth has slown down considerably.

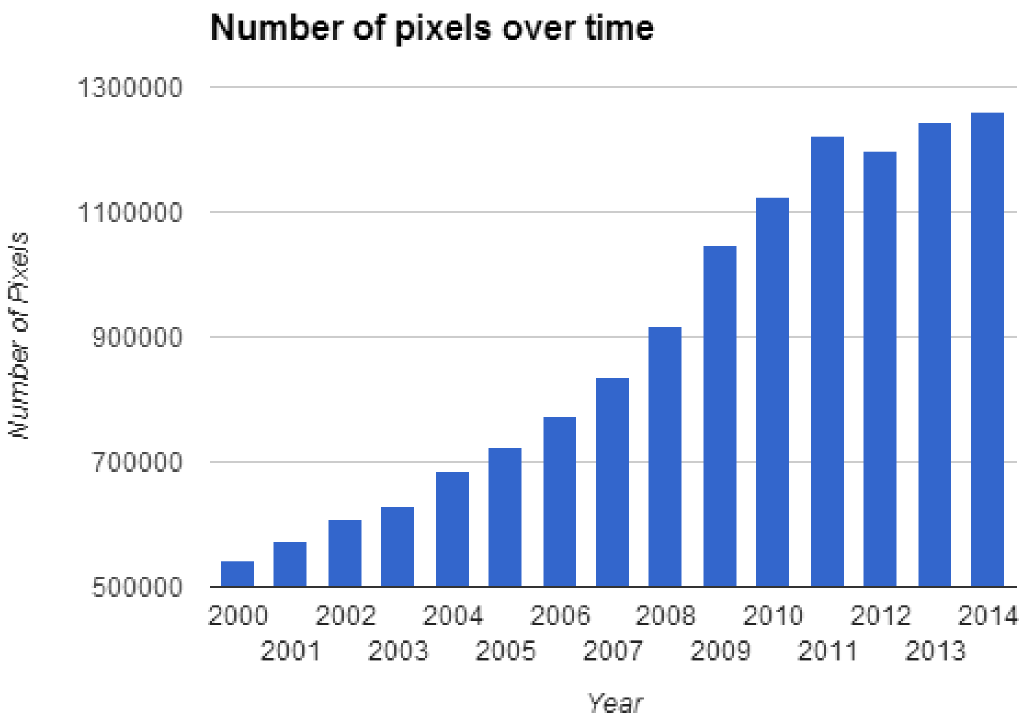
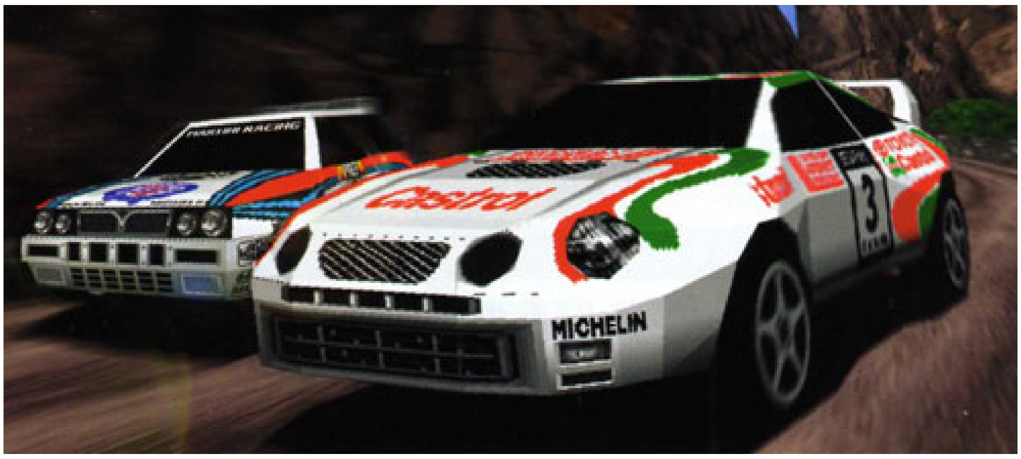
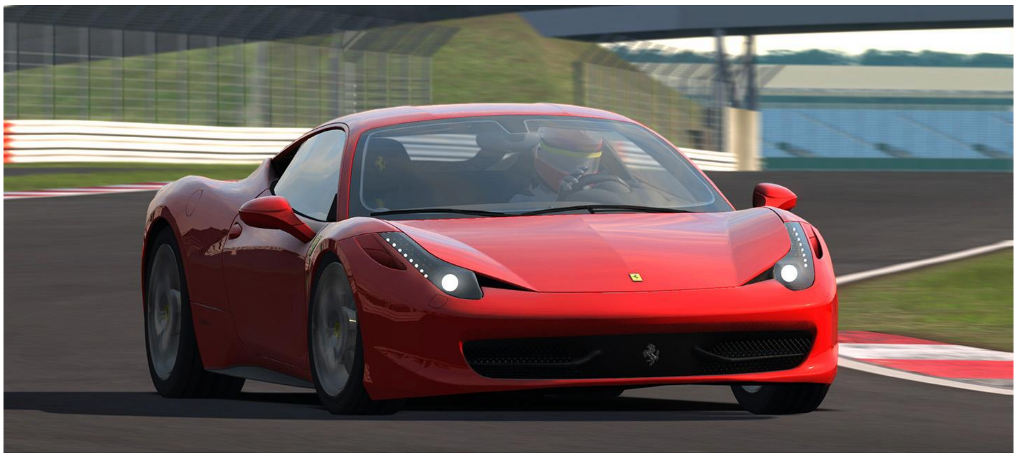


Chart generated using approximated statistics from w3school.com[9]

The number of triangles used in rendering has also grown considerably, and although there are no concrete statistics for this, we can compare two video games from the same genre from different times:



Sega Rally Championship[35] (1995). Approximate triangles per car: 200



Assetto Corsa[36] (2014). Approximate triangles per car: 55,000 - 65,000[34]

8. Conclusion

As shown by the results and from the differences between rasterisation and raytracing, it is obvious that on current hardware, raytracing is nowhere near as fast as rasterisation, simply because the hardware is not built to perform such tasks.

It is also notable that the worlds our games and movies inhabit have also been growing considerably, with no sign of slowing down, lending to even more triangles needing to be drawn.

As long as the number of triangles keeps growing, while the number of pixels on our screens stays the same, a time point where raytracing becomes more viable than rasterisation seems unavoidable, purely because of the way they scale.

It is therefore reasonable to conclude that due to the trends in the growing number of triangles and the slowing in screen size increases, that rasterisation is slowly becoming less efficient, while raytracing is becoming more viable for real-time rendering.

Although not currently as valuable, as hardware continually evolves, raytracing may have to become the new standard for real-time rendering, some time in the future.

References

[1] Python. http://www.python.org

[2] pygame. http://www.pygame.org

[3] PyOpenGL. http://pyopengl.sourceforge.net/

[4] PyOpenCL. http://mathema.tician.de/software/pyopencl/

[5] OpenGL. http://www.opengl.org

[6] DirectX. http://www.gamesforwindows.com/en-US/directx/

[7] OpenCL. https://www.khronos.org/opencl

[8] CUDA. http://www.nvidia.com/object/cuda_home_new.html

[9] Browser display statistics.
http://www.w3schools.com/browsers/browsers_display.asp

[10] Battlefield Series. http://www.battlefield.com

[11] Jason L. McKesson. Learning Modern 3d Graphics Programming.
http://pages.cs.wisc.edu/~mick/downloads/TutorialsComp.pdf, 27 April 2014.

[12] The Khronos Group. OpenCL Specification.
http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf, 27 April 2014.

[13] The Khronos Group. OpenGL Documentation.
http://www.opengl.org/sdk/docs/manglsl/, 27 April 2014.

[14] The Khronos Group. OpenGL API Reference Card.
http://www.khronos.org/files/opengl42-quick-reference-card.pdf, 27 April 2014.

[15] Dean Calver. Beyond3D. Photo-realistic Deferred Rendering.
http://www.beyond3d.com/content/articles/19/, 27 April 2014.

[16] Hansong Zhang. Forward Shadow Mapping. TR98-003. Department of Computer Science, University of North Carolina.
http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.48.2200, 27 April 2014.

[17] John Amanatides and Kin Choi. Ray Tracing Triangular Meshes. M3J 1P3. Dept. of Computer Science, York University.
http://www.cse.yorku.ca/~amana/research/mesh.pdf, 27 April 2014.

[18] Jaap Suter. Introduction to Octrees.
http://www.flipcode.com/archives/Introduction_To_Octrees.shtml, 27 April 2014.

[19] Austin Netinson. NVIDIA. Interactive Raytracing on the GPU and NVIRT overview. http://graphics.cs.williams.edu/i3d09/NVIRT-Overview.pdf, 27 April 2014.

[20] Marco Amalia. Deferred Rendering Shadow Mapping.
http://www.codinglabs.net/tutorial_opengl_deferred_rendering_shadow_mapping.aspx, 27 April 2014.

[21] Marco Amalia. Simple OpenGL Deferred Rendering.
http://www.codinglabs.net/tutorial_simple_def_rendering.aspx, 27 April 2014.

[22] enj. Adventures in PyOpenCL Part 1.
http://enja.org/2011/02/22/adventures-in-pyopencl-part-1-getting-started-with-python/, 27 April 2014.

[23] Lighthouse3d. GLSL 1.2 Tutorial.
http://www.lighthouse3d.com/tutorials/glsl-tutorial/, 27 April 2014.

[24] David Wolff. The Basics of GLSL 4.0 Shaders.
http://www.gamedev.net/page/resources/_/technical/opengl/the-basics-of-glsl-40-shaders-r2861, 27 April 2014.

[25] Gaston Hiller. Easy OpenCL with Python.
http://www.drdobbs.com/open-source/easy-opencl-with-python/240162614, 27 April 2014.

[26] Samuli Laine and Tero Karras. NVIDIA. Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation.
http://www.nvidia.com/docs/IO/88972/nvr-2010-001.pdf, 27 April 2014.

[27] GameStar. Raytracing vs Rasterisation.
http://www.scarydevil.com/~peter/io/raytracing-vs-rasterisation.html, 27 April 2014.

[28] Brent Owens. Forward Rendering vs Deferred Rendering.
http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342, 27 April 2014.

[29] Inkscape. http://inkscape.org/

[30] Gimp (Gnu Image Manipulation Program). http://www.gimp.org/

[31] Blender. http://www.blender.org/

[32] Thomas Diewald. Space Partitioning: Octree vs. BVH.
http://thomasdiewald.com/blog/?p=1488, 5 July 2014.

[33] Wikipedia. List of common 3D test models.
http://en.wikipedia.org/wiki/List_of_common_3D_test_models, 5 July 2014.

[34] Assetto Corsa Forms . Car Models for Assetto Corsa.
http://www.racedepartment.com/threads/car-models-for-assetto-corsa.53795/, 12 July 2014.

[35] Sega. Sega Rally Championship.
http://www.arcade-museum.com/game_detail.php?game_id=9475, 12 July 2014.

[36] Assetto Corsa. http://www.assettocorsa.net, 12 July 2014.

[37] 3D Julia Fractal. http://blog.hvidtfeldts.net/media/quatx.jpg, 23 July 2014.

Appendix: Implemented Algorithms

1. Raytracer.py

```python
from __future__ import division

import pygame
import os
import common
from OpenGL.GL import *
from OpenGL.GLU import *
try:
    from OpenGL import GLX
    print "Using X (Unix) window system"
except:
    from OpenGL import WGL
    print "Using Wiggle (windows) window system"
import numpy
import ctypes
from ctypes import *
#import PyOpenCL Objects
from pyopencl import Buffer, Program, Context, CommandQueue, GLTexture
#import PyOpenCL enumberations
from pyopencl import mem_flags, context_properties, platform_info
#import dtypes
from pyopencl.array import vec as cltypes
from pyopencl import tools as cltools
import pyopencl as OpenCL
import pyopencl.array as cl_array
from common.pyopengl import *
import itertools

from common.math3d import Vector

"""
Structures
"""

cltypes.Vertex = numpy.dtype([("position", cltypes.float4),
                              ("normal", cltypes.float4)])
cltypes.Camera = numpy.dtype([("position", cltypes.float4),
                              ("forward", cltypes.float4),
                              ("up", cltypes.float4),
                              ("right", cltypes.float4)])
cltypes.OctreeInfo = numpy.dtype([("size", numpy.float32)])

"""
EXTENSION METHODS
"""
#Set renderer specific object methods. Makes code cleaner

#Extension method of Device class
#Returns wether device meets renderer requirements
def _pyopencl_Device__meets_requirements(self):
    #TODO: Maybe
    return "cl_khr_gl_sharing" in self.extensions
#Apply extension method
OpenCL.Device.meets_requirements = _pyopencl_Device__meets_requirements

#Extension method for Camera objects
#Sets projection matrix
def _Camera__getCl_info(self):
    mat = self.rotation.matrix
    pos = self.position
    forward = mat*common.math3d.Vector(0, 0, 1)
    up = mat*common.math3d.Vector(0, 1, 0)
    right = mat*common.math3d.Vector(1, 0, 0)
    out = numpy.array(list(pos) + [0] +
                      list(forward) + [0] +
                      list(up) + [0] +
                      list(right) + [0],
                      dtype=numpy.float32)
    return numpy.ndarray((4, 4), numpy.float32, out)
#Apply extension method
common.objects.Camera.getCl_info = _Camera__getCl_info

#Extension method for Object objects
#Sets transformation matrix
def _Object__get_matrix(self):
    #calculate matrix from rotation, scale and translation
    rt = self.rotation.matrix
    return [rt[0], rt[1], rt[2], self.position[0],
            rt[3], rt[4], rt[5], self.position[1],
            rt[6], rt[7], rt[8], self.position[2],
            0,     0,     0,     1]
#Apply extension method
common.objects.Object.get_matrix = _Object__get_matrix

"""
   MAIN CLASS
"""

class Raytracer:
    """Raytraced Renderer"""

    """
       INITIALISATION
    """
    def __init__(self, resolution, object):
        #initialise display
        self.set_display(resolution)
        self.width, self.height = resolution

        #initialise opengl
        self.set_opengl()

        #initialise opencl
        self.set_opencl()

        #create the texture we render to
        self.create_texture()

        #Load object data
        self.object = object
        self.load_object()
        #Build octree
        self.generate_octree()

        #build program
        self.load_program()

        #print OpenGL Version
        print "Using OpenGL version: " + glGetString(GL_VERSION)

    def set_display(self, resolution):
        #Create a pygame window
        pygame.display.init()
        pygame.display.set_mode(resolution,
                                pygame.OPENGL|pygame.DOUBLEBUF)

    def set_opengl(self):
        #create shader program
        vertex_shader = Shader("raytraced/vertex.vert")
        fragment_shader = Shader("raytraced/fragment.frag")
        self.draw_program = ShaderProgram(vertex_shader.id, fragment_shader.id)

        #create render quad
        self.render_quad = QUAD()

    def set_opencl(self):
        #Get all devices that fit requirements
        #from one platform
        good_devices = []
        good_platform = None
        for platform in OpenCL.get_platforms():
            for device in platform.get_devices():
                if device.meets_requirements():
                    good_devices.append(device)
            if len(good_devices) > 0:
                good_platform = platform
                break

        #Raise a not supported exception if there are no good devices
        if len(good_devices) == 0:
            raise Exception("This program is not supported on your hardware")

        #Create a OpenCL context with platform specific properties
        properties = self.get_context_properties(good_platform)
        self.context = Context(good_devices, properties=properties)

        #Create the context queue
        self.queue = CommandQueue(self.context)

        #print OpenCL version
        print "Using OpenCL version: " + str(good_platform.get_info(platform_info.VERSION))

    def get_context_properties(self, plat):
        #reference context properties enumeration
        out = []

        #link OpenCL context platform
        out.append((context_properties.PLATFORM, plat))
        #link OpenGL context
        out.append((context_properties.GL_CONTEXT_KHR, platform.GetCurrentContext()))
        #link platform specific window contexts
        if "GLX" in globals():
            out.append((context_properties.GLX_DISPLAY_KHR, GLX.glXGetCurrentDisplay()))
        if "WGL" in globals():
            out.append((context_properties.WGL_HDC_KHR, WGL.wglGetCurrentDC()))

        #return context properties
        return out

    def load_program(self):
        #Read all the lines of the cl file into one string (safely)
        with open("raytraced/Raytracer.cl", "r") as file:
            source = file.read()
        octree_depth = self.octree.get_depth()
        source = source.replace("__REPLACE_OCTREE_DEPTH", str(octree_depth))

        #Create the opencl program
        program = Program(self.context, source)

        #make program options
        options = "-cl-mad-enable -cl-fast-relaxed-math -Werror -I %s" % os.path.dirname(os.path.abspath(__file__))

        #build program
        program.build(options=options)
        self.kernel = program.raytrace
        #Match argument types
        self.kernel.set_scalar_arg_dtypes([None, cltypes.Camera,
                                           None, numpy.int32,
                                           None, cltypes.OctreeInfo])

        #Match OpenCL Dtype. May not work everywhere
        cltypes.Vertex, c_decl = OpenCL.tools.match_dtype_to_c_struct(self.context.devices[0], 'Vertex', cltypes.Vertex)

    def create_texture(self):
        #Grab the screen size
        width, height = self.width, self.height

        #Create a GL texture objects
        self.gl_texture = texture = glGenTextures(1)

        #Bind the texture so we can change things
        glBindTexture(GL_TEXTURE_2D, texture)

        #Add parameters for texture access
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)

        #upload texture to GPU
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                     width, height, 0, GL_RGBA,
                     GL_FLOAT, None)

        #wait for OpenGL to finish executing every command
        glFinish()

        #Link the opengl texture to opencl
        self.render_texture = GLTexture(self.context,
                                        mem_flags.READ_WRITE,
                                        GL_TEXTURE_2D, 0,
                                        texture, 2)

    def load_object(self):
        vertices = []
        for tri in self.object.triangles:
            #Built vertex object
            position = tuple(list(self.object.vertices[tri]) + [0.0])
            normal = tuple(list(self.object.normals[tri]) + [0.0])
            vertex = (position, normal)
            vertices.append(vertex)

        self.meshes_array = numpy.array(vertices, dtype=cltypes.Vertex)
        print len(self.meshes_array) / 3
```

```
        #Make buffer
        self.meshes_buffer = Buffer(self.context, mem_flags.READ_ONLY | mem_flags.COPY_HOST_PTR,
hostbuf=self.meshes_array)

    def generate_octree(self):
        #Get the size of the octree
        size = 1
        for vertex in self.object.vertices:
            for value in vertex:
                size = max(abs(value), size)

        self.octree = Octree(size)
        for triangle in xrange(len(self.object.triangles) // 3):
            vertices = self.object.vertices[triangle*3:triangle*3 + 3]
            self.octree.add_triangle(vertices, triangle)

        self.octree_info = numpy.array(self.octree.size, dtype=numpy.float32)

        flattened = self.octree.flatten()
        self.octree_array = numpy.array(flattened, dtype=numpy.int32)

        #Make buffer
        self.octree_buffer = Buffer(self.context, mem_flags.READ_ONLY | mem_flags.COPY_HOST_PTR,
hostbuf=self.octree_array)

    def render(self, camera):
        camera_info = camera.getCl_info()

        #wait for OpenGL to finish all functions
        glFinish()
        #Bind OpenGL texture for OpenCL
        OpenCL.enqueue_acquire_gl_objects(self.queue, [self.render_texture])

        #Queue Raytrace
        self.raytrace(camera_info)

        #Unbind OpenGL texture from OpenCL
        OpenCL.enqueue_release_gl_objects(self.queue, [self.render_texture])

        #Render rendered texture to back-buffer
        self.render_render_texture()

        #Swap back and front buffers
        pygame.display.flip()

    def raytrace(self, camera_info):
        #Grab the global memory size (screen size)
        global_size = (self.width, self.height)

        #Execute OpenCL kernel with arguments
        self.kernel(self.queue, global_size, None,
                    #kernel arguments
                    self.render_texture, camera_info,
                    self.meshes_buffer, len(self.meshes_array),
                    self.octree_buffer, self.octree_info)

        #Wait for OpenCL to finish rendering
        self.queue.finish()

    def render_render_texture(self):
        #Bind shader program
        glUseProgram(self.draw_program.id)

        #Bind render-texture for reading
        glActiveTexture(GL_TEXTURE0 + self.gl_texture)
        glBindTexture(GL_TEXTURE_2D, self.gl_texture)
        #Set argument in shader program
        glUniform1i(self.draw_program.renderTexture, self.gl_texture)

        #Render render-quad with texture to back-buffer
        glCallList(self.render_quad)

    def close(self):
        #close the pygame window
        pygame.quit()
        #The rest should be handled by the OS and the hardware driver


OCTREE_CORNERS = (
    Vector(1, 1, 1), Vector(1, 1, -1), Vector(1, -1, -1), Vector(1, -1, 1),
    Vector(-1, 1, 1), Vector(-1, 1, -1), Vector(-1, -1, -1), Vector(-1, -1, 1),
)

class Node:
    def __init__(self, depth, origin, size):
        self.depth = depth
        self.origin = origin
        self.size = size
        self.children = [None] * 8
        self.triangles = []

    def add_triangle(self, mins, maxs, identifier):
        ch_size = self.size / 2
        #Test if triangle fits in children
        for index in xrange(len(self.children)):
            #Magic
            origin = self.origin + OCTREE_CORNERS[index] * ch_size
            offset = Vector(1, 1, 1) * ch_size
            lower = origin - offset
            upper = origin + offset

            if cube_contains_cube(lower, upper, mins, maxs):
                #Add child only if nessecary
                if self.children[index] is None:
                    self.children[index] = Node(self.depth + 1, origin, ch_size)
                #Recurse
                self.children[index].add_triangle(mins, maxs, identifier)
                return

        #Otherwise, we contain the triangle
        self.triangles.append(identifier)

    #FORMAT: children indecies (x8), length of contained triangles, *indexcies of contained triangles
    def flatten(self, offset = 0, parent = 0):
        flat = []
        #Add child references default 0
        for child in self.children:
            flat.append(0)
        #Add triangles
        flat.append(len(self.triangles))
        for triangle in self.triangles:
            flat.append(triangle)

        #Add child nodes and set their indecies
        for index in xrange(len(self.children)):
            child = self.children[index]
            if child is None:
                continue

            flat[index] = offset + len(flat)
            flat += child.flatten(flat[index], offset)
        return flat

    def __repr__(self):
        return "%s - %s" % (self.origin, self.size)

class Octree:
    def __init__(self, size, max_depth = 7):
        self.buffer = None
        self.max_depth = max_depth
        self.size = size
        self.root = Node(1, Vector(0, 0, 0), size)

    def flatten(self):
        return self.root.flatten()

    def get_depth(self, node=None):
        depth = 0
        nodes = [self.root]
        while nodes:
            node = nodes.pop()
            depth = max(node.depth, depth)
            for child in node.children:
                if child is not None:
                    nodes.append(child)
        return depth

    def add_triangle(self, vertices, identifier):
        mins = Vector(min(vertex[i] for vertex in vertices) for i in xrange(3))
        maxs = Vector(max(vertex[i] for vertex in vertices) for i in xrange(3))

        #Assume triangle is in the octree
        #Add the triangle recursively
        self.root.add_triangle(mins, maxs, identifier)

def cube_contains_cube(cube1_min, cube1_max, cube2_min, cube2_max):
    if False not in [cube1_min[i] <= cube2_min[i] for i in xrange(3)]:
        if False not in [cube1_max[i] >= cube2_max[i] for i in xrange(3)]:
            return True
    return False
```

2. Raytracer.cl

```
#include <Math.cl>

#define float4_ONE (float4)(1, 1, 1, 0)

__constant float4 OCTREE_CORNERS[8] = {
    (float4)(1, 1, 1, 0), (float4)(1, 1, -1, 0), (float4)(1, -1, -1, 0), (float4)(1, -1, 1, 0),
    (float4)(-1, 1, 1, 0), (float4)(-1, 1, -1, 0), (float4)(-1, -1, -1, 0), (float4)(-1, -1, 1, 0)
};

#define MAX_STACK_DEPTH __REPLACE_OCTREE_DEPTH

typedef struct {
    int node;
    int progress;
    bool hit;
    float node_size;
    float4 node_origin;
} OctreeStack;

typedef struct {
    float size;
} OctreeInfo;

//A vertex is made up of a position, a normal and a UV coordinate
typedef struct {
    float4 position;
    float4 normal;
} Vertex;

//A camera is made up of it's position and it's orientation
typedef struct {
    float4 position;
    float4 forward;
    float4 up;
    float4 right;
} Camera;

//A Ray-Hit point is made up of it's intersection point and information about it.
typedef struct {
    float4 point;
    float4 normal;
    float4 bary;
    float dist;
} Hit;

void distance_from_ray_to_plane(Ray *ray, float4 A, float4 B, float4 C, Hit *hit);
bool ray_triangle_check(Ray *ray, float4 A, float4 B, float4 C, Hit *hit);
void raycast_triangle_intersect(Ray *ray, const __global Vertex *vertices, int vertices_len, int triangle_index, Hit
*hit);

bool raycast(Ray *ray, const __global Vertex *vertices, int vertices_len,
             const __global int *octree, OctreeInfo octree_info, Hit *hit) {
    hit->dist = INFINITY;

    //Create stacks
    int depth = 0; //start at the top of the stack
    OctreeStack stack[MAX_STACK_DEPTH]; //The stack structure

    //Set the initial stack state
    stack[0].node = 0;
    stack[0].hit = false;
    stack[0].progress = 0;
    stack[0].node_size = octree_info.size + FLT_EPSILON;
    stack[0].node_origin = (float4)(0);

    while (depth >= 0) {
        if (stack[depth].hit) {
            //Check if this stack is completed
            if (stack[depth].progress == 8) {
                //go up
                depth--;
            }
            //Otherwise go to the next child
            else {
                while (stack[depth].progress < 8) {
                    int node_progress = stack[depth].progress;
                    int node_index = octree[stack[depth].node + node_progress];
                    //Always move on
                    stack[depth].progress++;
                    //Move down to child node
                    if (node_index != 0) {
                        depth++;
                        //Set up new stack
                        stack[depth].node = node_index;
                        stack[depth].hit = false;
                        stack[depth].progress = 0;
                        //calculate next origin
                        stack[depth].node_size = stack[depth - 1].node_size / 2;
                        float4 origin = stack[depth - 1].node_origin + OCTREE_CORNERS[node_progress] *
stack[depth].node_size;
                        stack[depth].node_origin = origin;
                        //dont do the rest yet
                        break;
                    }
                }
            }
        } else {
            // Test for collision
            float4 offset = float4_ONE * stack[depth].node_size;
            float4 box_min_coords = stack[depth].node_origin - offset;
            float4 box_max_coords = stack[depth].node_origin + offset;

            if (ray_box_intersection(ray, box_min_coords, box_max_coords)) {

                //intersect with triangles
                int current_node = stack[depth].node;
                int triangle_count = octree[current_node + 8];
                for (int index = 0; index < triangle_count; index++) {
                    //Do intersection test
                    int triangle_index = octree[current_node + 9 + index];
                    raycast_triangle_intersect(ray, vertices, vertices_len, triangle_index, hit);
                }
                //if (depth == 1) {
                //    return true;
                //}

                //mark as done
                stack[depth].hit = true;
            }
            //otherwise go up
            else {
                depth--;
            }
        }
    }

    return !(isinf(hit->dist));
}

void raycast_triangle_intersect(Ray *ray, const __global Vertex *vertices,
                                int vertices_len, int triangle_index, Hit *hit) {
    //Get vertex coordiates of every triangle
    float4 A = vertices[triangle_index * 3 + 0].position;
    float4 B = vertices[triangle_index * 3 + 1].position;
    float4 C = vertices[triangle_index * 3 + 2].position;

    //Get the hitdistace from the plane A, B, C
    Hit plane_hit;
    distance_from_ray_to_plane(ray, A, B, C, &plane_hit);

    if (plane_hit.dist == INFINITY) {
      return;
    }

    if(0 < plane_hit.dist && plane_hit.dist < hit->dist) {
        if (ray_triangle_check(ray, A, B, C, &plane_hit)) {
            *hit = plane_hit;
        }
    }
}

void distance_from_ray_to_plane(Ray *ray, float4 A, float4 B, float4 C, Hit *hit) {
    hit->normal = cross(C - A, B - A);
    float dotDirection = dot(hit->normal, ray->direction);

    if (dotDirection <= 0) {
        hit->dist = INFINITY;
        return;
    }

    hit->dist = dot(A - ray->origin, hit->normal) / dotDirection;
}
```

```
//Check whether an intersection point is inside the triangle A, B, C
//Intersection point is dist along the ray.
bool ray_triangle_check(Ray *ray, float4 A, float4 B, float4 C, Hit *hit) {
    hit->point = ray->origin + ray->direction  *hit->dist;

    float area = length(hit->normal);

    //Optimisation
    float4 QA = A - hit->point;
    float4 QB = B - hit->point;
    float4 QC = C - hit->point;

    //Barycentric intersection test
    hit->bary.x = length(cross(QB, QC))/area;
    hit->bary.y = length(cross(QA, QC))/area;
    hit->bary.z = length(cross(QA, QB))/area;

    hit->bary.w = hit->bary.x + hit->bary.y + hit->bary.z;

    //Check for bary intersection
    float diff = fabs(1.0 - hit->bary.w);
    if (diff < 0.000001) {
        return true;
    }

    return false;
}

__kernel void raytrace(__write_only image2d_t renderTexture, Camera camera,
                       const __global Vertex *vertices, int vertices_len,
                       const __global int *octree, OctreeInfo octree_info) {

    int x = get_global_id(0);
    int y = get_global_id(1);
    int2 position = (int2)(x, y);

    float width  = get_global_size(0);
    float height = get_global_size(1);
    //uv coordinates

    float normalised_x = x/width - 0.5;
    float normalised_y = y/height - 0.5;

    float2 normalised_pos = (float2)(normalised_x, normalised_y);

    Ray ray;
    ray.origin = camera.position;
    ray.direction = camera.forward + normalised_y*camera.up + normalised_x*camera.right;

    Hit hit;
    float4 color = BLACK;

    //Do raytracing
    if (raycast(&ray, vertices, vertices_len, octree, octree_info, &hit)) {
        float4 normal = normalize(hit.normal);
        color = dot(normal, ray.direction) * WHITE + WHITE * 0.4;
    }

    write_imagef(renderTexture, position, color);
}
```

3. Math.cl

```
#define BLACK (float4)(0, 0, 0, 1)
#define WHITE (float4)(1, 1, 1, 1)

typedef struct {
    float4 origin;
    float4 direction;
} Ray;

bool ray_box_intersection(Ray *ray, float4 box_min_coords, float4 box_max_coords) {
    //Magic ray-box intersection test (AABB)
    float inv_x = 1/ray->direction.x;
    float inv_y = 1/ray->direction.y;
    float inv_z = 1/ray->direction.z;

    //Some t intersections
    float t1 = (box_min_coords.x - ray->origin.x) * inv_x;
    float t2 = (box_max_coords.x - ray->origin.x) * inv_x;
    float t3 = (box_min_coords.y - ray->origin.y) * inv_y;
    float t4 = (box_max_coords.y - ray->origin.y) * inv_y;
    float t5 = (box_min_coords.z - ray->origin.z) * inv_z;
    float t6 = (box_max_coords.z - ray->origin.z) * inv_z;

    float tmax = min(min(max(t1, t2), max(t3, t4)), max(t5, t6));

    //box is behind ray
    if (tmax < 0) {
        return false;
    }

    float tmin = max(max(min(t1, t2), min(t3, t4)), min(t5, t6)) - 0.0001;

    //ray does not intersect
    if (tmin > tmax) {
        return false;
    }

    return true;
}
```

4. Rasteriser.py

```
import pygame
import OpenGL
import common.objects
from OpenGL.GL import *
from OpenGL.GL.ARB.framebuffer_object import *
from OpenGL.GLU import *
from common.pyopengl import *

"""
   EXTENTION METHODS
"""
#Set renderer specific object methods. Makes code cleaner

#Extension method for Mesh objects
#Generates a OpenGL List Object for the mesh
def _Mesh__generate_glList(self):
    #Create a new list object
    self.listid = id = glGenLists(1)
    #Start defining a list object
    glNewList(id, GL_COMPILE)

    #Begin adding triangle data
    glBegin(GL_TRIANGLES)
    for tris in self.triangles:
        #add normal, uv and vertex data
        if len(self.normals) != 0:
            glNormal3f(*self.normals[tris])
        glVertex3f(*self.vertices[tris])
    #End adding triangle data
    glEnd()

    #end defining a list object
    glEndList()
#Apply extension method
common.objects.Mesh.generate_glList = _Mesh__generate_glList

#Extension method for Camera objects
#Sets projection matrix
def _Camera__apply_glMatrix(self):
    #calculate aspect ratio
    width, height = pygame.display.get_surface().get_size()

    glLoadIdentity()
    #Set perspective matrix
    gluPerspective(self.fov, float(width)/float(height), self.near, self.far)

    #Apply rotation matrix
    rt = self.rotation.matrix
    matrix = [rt[0], rt[1], rt[2], 0,
              rt[3], rt[4], rt[5], 0,
              rt[6], rt[7], rt[8], 0,
              0,     0,     0,     1]
    glMultMatrixf(matrix)
    glScalef(1, 1, -1)
    #Apply translation
    glTranslatef(*-self.position)
#Apply extension method
common.objects.Camera.apply_glMatrix = _Camera__apply_glMatrix

"""
   MAIN CLASS
```

```
"""

class Rasteriser:
    """Deferred Rastoriser"""

    """
       INITIALIZATION
    """
    def __init__(self, resolution, object):
        #Setup the pygame screen
        self.set_display(resolution)

        #set opengl global parameters
        self.set_opengl()
        #load glsl shaders
        self.load_shaders()

        #Add object
        object.generate_glList()
        self.object = object

        #Print OpenGL version
        print "Using OpenGL version: " + glGetString(GL_VERSION)

    def set_display(self, resolution):
        #initialize pygame display and set the pygame display mode appropriately
        pygame.display.init()
        pygame.display.set_mode(resolution, pygame.OPENGL|pygame.DOUBLEBUF)

    def set_opengl(self):
        glEnable(GL_DEPTH_TEST) #Z Buffer
        glEnable(GL_CULL_FACE) #face culling
        glClearColor(0.0, 0.0, 0.0, 0.0)

    def load_shaders(self):
        #Load shader objects from Hardcoded shader paths
        vertex = Shader("rasterised/shader.vert")
        fragment = Shader("rasterised/shader.frag")

        #Create program objects
        self.shader = ShaderProgram(vertex.id, fragment.id)
        glUseProgram(self.shader.id)

    """
       RUNTIME
    """

    #render with camera
    def render(self, camera):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

        #Setup camera projection matrix
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        camera.apply_glMatrix()

        #Draw object
        glCallList(self.object.listid)

        #Flip front and back buffers
        pygame.display.flip()

    """
       CLEANUP
    """

    def close(self):
        #cleanup after ourselves
        pygame.quit()
        #Drivers should be smart enough to clean up the mess we left
```

5. shader.frag

```
varying vec3 position;
varying vec3 normal;

void main(void) {
    float shading = dot(normalize(position), normalize(normal));
    gl_FragColor = vec4(vec3(shading), 1.0);
}
```

6. shader.vert

```
varying vec3 position;
varying vec3 normal;

void main(void) {
    normal = gl_NormalMatrix * gl_Normal;

    position = vec3(gl_Vertex * gl_ProjectionMatrix);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

7. ObjImporter.py

```
"""
Contains functions for loading optimised Wavefront .obj
files into nicely formatted objects with optimised data.
"""


import string
from math3d import Vector
from objects import Mesh

def load(path):
    """
    load(path:str) -> [Mesh,]

    Loads a .obj file from "path"
    and returns it in a Mesh object
    """

    objects = []

    with open(path, "r") as file:
        lines = file.readlines()
        objs = __split_objects(lines)
        for obj in objs:
            obj = __parse_lines(obj)
            if obj:
                obj.file = path
                objects.append(obj)

    return objects

def __split_objects(lines):
    objects = []

    index = 0
    last_index = 0
    while index < len(lines):
        values = lines[index].split(" ")
        index += 1

        if values[0] == "o":
            objects.append(lines[last_index:index])
            last_index = index
    objects.append(lines[last_index:])

    return objects

def __parse_lines(lines):
    verts = []
    norms = []
    tris = []

    for line in lines:
        attributes = line.split(" ")

        if attributes[0] == "#" or len(attributes) < 2:
            continue

        if attributes[0] == "v":
            verts.append(Vector(
                float(attributes[1]),
                float(attributes[2]),
                float(attributes[3])));
        elif attributes[0] == "vn":
            norms.append(Vector(
                float(attributes[1]),
                float(attributes[2]),
                float(attributes[3])));
        elif attributes[0] == "f":
            i = 3
            while i < len(attributes):
                for j in xrange(3):
                    values = attributes[i - 2 + j].split("/")
                    value = Vector(-1, -1, -1)
                    value[0] = int(values[0]) - 1
                    if len(values) == 3:
                        if values[1] != "":
                            value[1] = int(values[1]) - 1
                        if values[2] != "":
                            value[2] = int(values[2]) - 1
                    tris.append(value)
                i += 1

    if len(tris) == 0:
        return

    vertices = []
    normals = []

    triangles = []

    index = 0
    for triangle in tris:
        vertices.append(verts[int(triangle[0])])
        normals.append(norms[int(triangle[2])])
        triangles.append(index)
        index += 1

    mesh = Mesh()
    mesh.vertices = tuple(vertices)
    mesh.normals = tuple(normals)
    mesh.triangles = tuple(triangles)

    return mesh
```

8. math3d.py

```
"""This module has definitions for some useful
mathematical values represented in pure python,
such as vector maths, quaternion and matrices
as well as some useful shortcut solutions
to some common problems

Works best with pypy (Not CPython)
"""

import math

class _classproperty(property):
    def __get__(self, cls, owner):
        return classmethod(self.fget).__get__(None, owner)()

class DimentionMissmatchException(Exception): pass

class Vector():
    """A Generalised Vector class deigned with minimal overhead.
    The Vectors dimension number is determined when it is created
    and stays constant for it's lifetime.
    """
    __slows__ = ["_value"]
    def __init__(self, *args):
        """Makes a new Vector with either
        each vector component separately
        or any iterable as arguments
        """
        if len(args) == 1:
            args = args[0]
        self._value = tuple(float(arg) for arg in args)

    def __len__(self):
        """Returns the dimension number"""
        return len(self._value)

    def __str__(self):
        """Returns a nicely formatted string representation of the Vector"""
        return str(self._value)
    def __repr__(self):
        """Returns a nicely formatted string representation of the Vector"""
        return repr(self._value)
```

```
def __add__(vec1, vec2):
    """Adds two Vectors with the same dimension number.
    Raises a DimentionMissmatchException if they don't match
    """
    if len(vec1) == len(vec2):
        return Vector(vec1[i] + vec2[i] for i in range(len(vec1)))
    raise DimentionMissmatchException()
__radd__ = __add__

def __iadd__(self, other):
    """Adds another Vector to itself.
    Raises a DimentionMissmatchException
    if the dimension numbers don't match
    """
    if len(self) == len(other):
        self._value = tuple(self[i] + other[i] for i in range(len(self)))
        return self
    raise DimentionMissmatchException()

def __sub__(vec1, vec2):
    """Subtracts two Vectors with the same dimension number.
    Raises a DimentionMissmatchException if they don't match
    """
    if len(vec1) == len(vec2):
        return Vector(vec1[i] - vec2[i] for i in range(len(vec1)))
    raise DimentionMissmatchException()
__rsub__ = __sub__

def __isub__(self, other):
    """Subtracts another Vector to itself.
    Raises a DimentionMissmatchException
    if the dimension numbers don't match
    """
    if len(self) == len(other):
        self._value = tuple(self[i] - other[i] for i in range(len(self)))
        return self
    raise DimentionMissmatchException()

def __neg__(self):
    """Returns the negative value of the Vector"""
    return Vector(-value for value in self)

def __mul__(vec, scalar):
    """Multiplies a Vector by a Scalar"""
    scalar = float(scalar)
    return Vector(value*scalar for value in vec)
__rmul__ = __mul__

def __imul__(self, other):
    """Multiplies another Scalar by itself"""
    other = float(other)
    self._value = tuple(self[i]*other for i in range(len(self)))

def __div__(vec, scalar):
    """Divides a Vector by a Scalar"""
    scalar = float(scalar)
    return Vector(value/scalar for value in vec)
__truediv__ = __div__

def __floordiv__(vec, scalar):
    """Divides a Vector by an integer Scalar"""
    scalar = int(scalar)
    return Vector(value/scalar for value in vec)

def __idiv__(self, other):
    """Divides another Scalar by itself"""
    other = float(other)
    self._value = tuple(self[i]/other for i in range(len(self)))

def __eq__(vec1, vec2):
    """Checks equality between Vectors"""
    if len(vec1) != len(vec2): return False

    return False not in [vec1[i] == vec2[i] for i in range(len(vec1))]

def __getitem__(self, key):
    """Get a value from a Vector given a dimension"""
    return self._value[key]

def __setitem__(self, key, value):
    """Set a value from a Vector given a dimension"""
    self._value = tuple(float(value) if i == key else self[i]
                        for i in range(len(self)))
```

```
def __iter__(self):
    """Returns the iterator of the internal Tuple object"""
    return iter(self._value)

def __list__(self):
    return list(self._value)

def __tuple__(self):
    return tuple(self._value)

def __float__(self):
    return self.magnitude

def __int__(self):
    return len(self)

@property
def x(self):
    """Shorthand for getting Vector[0]"""
    return self[0]

@x.setter
def x(self, value):
    """Shorthand for setting Vector[0]"""
    self[0] = value

@property
def y(self):
    """Shorthand for getting Vector[1]"""
    return self[1]

@y.setter
def y(self, value):
    """Shorthand for setting Vector[1]"""
    self[0]= value

@property
def z(self):
    """Shorthand for getting Vector[2]"""
    return self[2]

@z.setter
def z(self, value):
    """Shorthand for setting Vector[2]"""
    self[2] = value

@property
def w(self):
    """Shorthand for getting Vector[3]"""
    return self[3]

@w.setter
def w(self, value):
    """Shorthand for setting Vector[3]"""
    self[3] = value

@property
def magnitude(self):
    """Returns the magnitude of the Vector"""
    return sum(value**2 for value in self)**0.5

@magnitude.setter
def magnitude(self, value):
    """Sets the magnitude of the Vector
    Direction is maintained
    """
    multi = float(value)/self.magnitude
    self._value = tuple(val*multi for val in self)

@property
def magnitude2(self):
    """Returns the magnitude squared of the Vector
    Useful for comparing lengths
    """
    return sum(value**2 for value in self)

@property
def normalized(self):
    """Returns the normalized direction Vector"""
    return self/self.magnitude

@normalized.setter
def normalized(self, value):
        """Sets the normalized direction Vector
        Magnitude is maintained
        """
        if len(self) == len(value):
            mag = self.magnitude
            self._value = tuple(val*mag for val in value)
        else:
            raise DimentionMissmatchException()

    def normalize(self):
        """Normalizes the Vector"""
        magnitude = self.magnitude
        self._value = tuple(value/magnitude for value in self)

    @classmethod
    def dot(cls, vect1, vect2):
        """Returns the dot product between two Vectors"""
        if len(vect1) == len(vect2):
            return sum(vect1[i]*vect2[i] for i in range((len(vect1))))
        raise DimentionMissmatchException()

    @classmethod
    def angle(cls, vect1, vect2):
        """Returns the angle between two Vectors in radians"""
        return math.acos(cls.dot(vect1, vect2)/
                         vect1.magnitude*vect2.magnitude)

    @classmethod
    def angleD(cls, vect1, vect2):
        """Returns the angle between two Vectors in degrees"""
        return cls.angle(vect1, vect2)*180/math.pi

    @classmethod
    def distance(cls, vect1, vect2):
        """Returns the distance between two Vectors"""
        return (vect1 - vect2).magnitude

    @classmethod
    def scale(cls, vect1, vect2):
        """Multiplies two Vectors component wise"""
        if len(vect1) == len(vect2):
            return Vector(vect1[i] * vect2[i] for i in range(len(vect1)))
        raise DimentionMissmatchException()

    @classmethod
    def cross2(cls, vect):
        """Returns the two dimensional cross product of a Vector.
        Ignores other dimensions
        """
        if len(vect) >= 2:
            return Vector(-vect[1], vect[0])
        raise DimentionMissmatchException()

    @classmethod
    def cross3(cls, v1, v2):
        """Returns the three dimensional cross product of two Vectors.
        Ignores other dimensions
        """
        if len(v1) == len(v2) >= 3:
            return Vector(v1[1]*v2[2] - v1[2]*v2[1],
                          v1[2]*v2[0] - v1[0]*v2[2],
                          v1[0]*v2[1] - v1[1]*v2[0])
        raise DimentionMissmatchException()

class Quaternion(object):
    __slots__ = ["_value"]
    def __init__(self, *args):
        if len(args) == 4:
            self._value = (float(args[0]), float(args[1]), float(args[2]), float(args[3]))
        elif len(args) == 1:
            args = args[0]
            self._value = (float(args[0]), float(args[1]), float(args[2]), float(args[3]))
        else:
            raise AttributeError()

    def __len__(self):
        return len(self._value)

    def __str__(self):
        return str(self._value)
    __repr__ = __str__

    def __mul__(self, other):
        if len(other) == len(self):
            return Quaternion(
                self[3]*other[3] - self[1]*other[1] - self[2]*other[2] - self[0]*other[0],
                self[3]*other[1] + self[1]*other[3] + self[2]*other[0] - self[0]*other[2],
                self[3]*other[2] - self[1]*other[0] + self[2]*other[3] + self[0]*other[1],
                self[3]*other[0] + self[1]*other[2] - self[2]*other[1] + self[0]*other[3]
            )
        return self.matrix*other
    __rmul__ = __mul__

    def __getitem__(self, key):
        return self._value[key]

    def __setitem__(self, key, value):
        values = list(self._value)
        values[key] = float(value)
        self._value = tuple(values)

    @property
    def x(self):
        return self[0]

    @x.setter
    def x(self, value):
        self._value = (float(value), self[1], self[2], self[3])

    @property
    def y(self):
        return self[1]

    @y.setter
    def y(self, value):
        self._value = (self[0], float(value), self[2], self[3])

    @property
    def z(self):
        return self[2]

    @z.setter
    def z(self, value):
        self._value = (self[0], self[1], float(value), self[3])

    @property
    def w(self):
        return self[3]

    @w.setter
    def w(self, value):
        self._value = (self[0], self[1], self[2], float(value))

    @property
    def matrix(self):
        return Matrix3x3(
            (1 - 2*self.y**2 - 2*self.z**2), 2*(self.x*self.y + self.w*self.z), 2*(self.x*self.z - self.w*self.y),
            2*(self.x*self.y - self.w*self.z), (1 - 2*self.x**2 - 2*self.z**2), 2*(self.y*self.z + self.w*self.x),
            2*(self.x*self.z + self.w*self.y), 2*(self.y*self.z - self.w*self.x), (1 - 2*self.x**2 - 2*self.y**2)
        )

    @classmethod
    def euler(cls, *args):
        if len(args) == 3:
            args = [Vector(*args)]
        angles = args[0]

        c1 = math.cos(angles.y/2)
        s1 = math.sin(angles.y/2)
        c2 = math.cos(angles.z/2)
        s2 = math.sin(angles.z/2)
        c3 = math.cos(angles.x/2)
        s3 = math.sin(angles.x/2)
        c1c2 = c1*c2
        s1s2 = s1*s2
        return Quaternion(
            c1c2*s3 + s1s2*c3,
            s1*c2*c3 + c1*s2*s3,
            c1*s2*c3 - s1*c2*s3,
            c1c2*c3 - s1s2*s3
        )

    @_classproperty
    def identity(cls):
        return Quaternion(0, 0, 0, 1)

class Matrix3x3(object):
    def __init__(self, *args):
        if len(args) == 9:
            self._value = tuple(args)
        else:
            raise AttributeError()

    def __len__(self):
        return (3, 3)

    def __getitem__(self, key):
        return self._value[key]

    def __setitem__(self, key, value):
        values = list(self._value)
        values[key] = float(value)
        self._value = tuple(values)

    def __mul__(self, other):
        if len(other) == 3:
            return Vector(
                other[0]*self[0] + other[1]*self[1] + other[2]*self[2],
                other[0]*self[3] + other[1]*self[4] + other[2]*self[5],
                other[0]*self[6] + other[1]*self[7] + other[2]*self[8])
        if len(other) == (3, 3):
            raise NotImplementedError()
```

9. pyopengl.py

```
"""
Holds classes for simpler opengl work
"""

from OpenGL.GL import *

class Shader():
    #Shader file extensions
    _vertex_extensions = ["vert", "v"]
    _fragment_extensions = ["frag", "f"]

    def __init__(self, path, type=None):
        #infer shader type from file extension
        if not type: type = Shader.get_type(path)

        #create shader object
        self.id = shader = glCreateShader(type)

        #compile shader from file
        with open(path) as file:
            lines = file.readlines()
        glShaderSource(shader, lines)
        glCompileShader(shader)

        #print shader log
        info = glGetShaderInfoLog(shader)
        if info: print info

    @classmethod
    def get_type(cls, path):
        #get file extension
        extension = path.split(".")[-1]

        #Map file extension to shader type
        if extension in cls._vertex_extensions:
            return GL_VERTEX_SHADER
        elif extension in cls._fragment_extensions:
            return GL_FRAGMENT_SHADER

        #throw error if type could not be inferred
        raise Exception("Could not infer Shader type from file extension")

class ShaderProgram():
    def __init__(self, vertex, fragment):
        #Create a shader program
        self.id = id = glCreateProgram()

        #Attach and upload shaders
        glAttachShader(id, vertex)
        glAttachShader(id, fragment)
        glLinkProgram(id)

        #Try using the shader. OpenGL will raise Exceptions if there was an error
        try:
            glUseProgram(id)
        except OpenGL.error.GLError:
            print glGetProgramInfoLog(id)
        glUseProgram(0)

        #Get active Uniform values from Shader Program and add them to object by name
        for i in range(glGetProgramiv(id, GL_ACTIVE_UNIFORMS)):
            #Try block in case of implementation error
            try:
                name = glGetActiveUniform(id, i)[0]
                #only add uniforms that are possible to access by python
                if "." not in name:
                    self.__dict__[name] = i
            except:
                continue

def QUAD():
    #Create a primitive Quad
    id = glGenLists(1)
    glNewList(id, GL_COMPILE)
    glBegin(GL_QUADS)
    glTexCoord2f(0, 0)
    glVertex2f(-1, -1)
    glTexCoord2f(1, 0)
    glVertex2f(1, -1)
    glTexCoord2f(1, 1)
    glVertex2f(1, 1)
    glTexCoord2f(0, 1)
    glVertex2f(-1, 1)
    glEnd()
    glEndList()
    return id
```

10. Raw Results

```
raytraced
 320x340 (108800 pixels)
  cube.obj: 61.9985609678fps with 12 triangles
  ico-sphere.obj: 60.451024923fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 10.0046874946fps with 29964 triangles
  stanford-bunny.obj: 29.9995499134fps with 5002 triangles
  suzanne-subdivision-1.obj: 50.4711091233fps with 3936 triangles
  suzanne-subdivision-2.obj: 18.1992381871fps with 15744 triangles
  suzanne.obj: 60.3203025992fps with 968 triangles
  torus.obj: 60.1562685685fps with 2048 triangles
  uv-sphere.obj: 60.2841476532fps with 960 triangles
 480x360 (172800 pixels)
  cube.obj: 61.323182446fps with 12 triangles
  ico-sphere.obj: 60.0231786645fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 4.62037401088fps with 29964 triangles
  stanford-bunny.obj: 17.1637723983fps with 5002 triangles
  suzanne-subdivision-1.obj: 20.0047462376fps with 3936 triangles
  suzanne-subdivision-2.obj: 7.0587881793fps with 15744 triangles
  suzanne.obj: 60.0343158151fps with 968 triangles
  torus.obj: 60.0183941438fps with 2048 triangles
  uv-sphere.obj: 60.1248843413fps with 960 triangles
 800x600 (480000 pixels)
  cube.obj: 61.2541663816fps with 12 triangles
  ico-sphere.obj: 27.5711216021fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 1.79896265609fps with 29964 triangles
  stanford-bunny.obj: 6.66312701633fps with 5002 triangles
  suzanne-subdivision-1.obj: 8.03128189479fps with 3936 triangles
  suzanne-subdivision-2.obj: 2.60687494572fps with 15744 triangles
  suzanne.obj: 20.034493784fps with 968 triangles
  torus.obj: 25.9819427954fps with 2048 triangles
  uv-sphere.obj: 41.5289512381fps with 960 triangles
 1024x768 (786432 pixels)
  cube.obj: 60.6787933295fps with 12 triangles
  ico-sphere.obj: 20.0125698394fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 1.46880390777fps with 29964 triangles
  stanford-bunny.obj: 5.11529105989fps with 5002 triangles
  suzanne-subdivision-1.obj: 6.67493482532fps with 3936 triangles
  suzanne-subdivision-2.obj: 2.30616326472fps with 15744 triangles
  suzanne.obj: 19.9987745196fps with 968 triangles
  torus.obj: 20.0042646111fps with 2048 triangles
  uv-sphere.obj: 30.0323463085fps with 960 triangles
 1280x800 (1024000 pixels)
  cube.obj: 60.8723918116fps with 12 triangles
  ico-sphere.obj: 19.9929979812fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 1.36271652193fps with 29964 triangles
  stanford-bunny.obj: 4.6192449742fps with 5002 triangles
  suzanne-subdivision-1.obj: 6.66553745319fps with 3936 triangles
  suzanne-subdivision-2.obj: 2.30180936788fps with 15744 triangles
  suzanne.obj: 15.031485056fps with 968 triangles
  torus.obj: 20.0125549406fps with 2048 triangles
  uv-sphere.obj: 30.0049188417fps with 960 triangles
 1280x1024 (1310720 pixels)
  cube.obj: 60.5942488746fps with 12 triangles
  ico-sphere.obj: 15.0077704814fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 1.07079754351fps with 29964 triangles
  stanford-bunny.obj: 3.57331374726fps with 5002 triangles
  suzanne-subdivision-1.obj: 5.00761364948fps with 3936 triangles
  suzanne-subdivision-2.obj: 1.7637033004fps with 15744 triangles
  suzanne.obj: 12.0096932673fps with 968 triangles
  torus.obj: 15.0121317967fps with 2048 triangles
  uv-sphere.obj: 20.0472108409fps with 960 triangles
 1366x768 (1049088 pixels)
  cube.obj: 60.8214113715fps with 12 triangles
  ico-sphere.obj: 20.0153384221fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 1.39536221435fps with 29964 triangles
  stanford-bunny.obj: 4.61596316822fps with 5002 triangles
  suzanne-subdivision-1.obj: 6.66467297584fps with 3936 triangles
  suzanne-subdivision-2.obj: 2.30863940748fps with 15744 triangles
  suzanne.obj: 18.2000292668fps with 968 triangles
  torus.obj: 20.011047137fps with 2048 triangles
  uv-sphere.obj: 30.0281748472fps with 960 triangles
 1920x1080 (2073600 pixels)
  cube.obj: 60.2066009364fps with 12 triangles
  ico-sphere.obj: 11.997731769fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 0.901934677605fps with 29964 triangles
  stanford-bunny.obj: 2.72713090008fps with 5002 triangles
  suzanne-subdivision-1.obj: 4.285288188fps with 3936 triangles
  suzanne-subdivision-2.obj: 1.66608463994fps with 15744 triangles
  suzanne.obj: 8.5837031696fps with 968 triangles
  torus.obj: 12.0073965069fps with 2048 triangles
  uv-sphere.obj: 20.0152429578fps with 960 triangles

rasterised
 320x340 (108800 pixels)
  cube.obj: 61.3317369777fps with 12 triangles
  ico-sphere.obj: 60.3820702601fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 61.8214928135fps with 29964 triangles
  stanford-bunny.obj: 62.7609422341fps with 5002 triangles
  suzanne-subdivision-1.obj: 60.3572098384fps with 3936 triangles
  suzanne-subdivision-2.obj: 61.8500586934fps with 15744 triangles
  suzanne.obj: 60.6776129064fps with 968 triangles
  torus.obj: 62.9274701123fps with 2048 triangles
  uv-sphere.obj: 63.1279723921fps with 960 triangles
 480x360 (172800 pixels)
  cube.obj: 63.6816101175fps with 12 triangles
  ico-sphere.obj: 62.5831721046fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 60.2702206175fps with 29964 triangles
  stanford-bunny.obj: 60.3602037968fps with 5002 triangles
  suzanne-subdivision-1.obj: 60.6752798684fps with 3936 triangles
  suzanne-subdivision-2.obj: 61.7322243083fps with 15744 triangles
  suzanne.obj: 62.6800795904fps with 968 triangles
  torus.obj: 62.9713968918fps with 2048 triangles
  uv-sphere.obj: 60.6672991545fps with 960 triangles
 800x600 (480000 pixels)
  cube.obj: 62.2309156459fps with 12 triangles
  ico-sphere.obj: 60.5883353875fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 60.3832246941fps with 29964 triangles
  stanford-bunny.obj: 62.6922044299fps with 5002 triangles
  suzanne-subdivision-1.obj: 60.3721962456fps with 3936 triangles
  suzanne-subdivision-2.obj: 61.5322786465fps with 15744 triangles
  suzanne.obj: 63.1323146923fps with 968 triangles
  torus.obj: 60.8814695869fps with 2048 triangles
  uv-sphere.obj: 63.5265151905fps with 960 triangles
 1024x768 (786432 pixels)
  cube.obj: 62.333875015fps with 12 triangles
  ico-sphere.obj: 61.0684787553fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 61.8306771312fps with 29964 triangles
  stanford-bunny.obj: 62.1760737481fps with 5002 triangles
  suzanne-subdivision-1.obj: 60.6809390038fps with 3936 triangles
  suzanne-subdivision-2.obj: 60.8712009183fps with 15744 triangles
  suzanne.obj: 60.7441542564fps with 968 triangles
  torus.obj: 63.175473259fps with 2048 triangles
  uv-sphere.obj: 60.6785752651fps with 960 triangles
 1280x800 (1024000 pixels)
  cube.obj: 62.3183574765fps with 12 triangles
  ico-sphere.obj: 60.7081657879fps with 1280 triangles
  stanford-bunny-subdivision-1.obj: 61.4367277587fps with 29964 triangles
  stanford-bunny.obj: 61.4930772621fps with 5002 triangles
  suzanne-subdivision-1.obj: 60.7292777498fps with 3936 triangles
  suzanne-subdivision-2.obj: 61.0017490083fps with 15744 triangles
  suzanne.obj: 60.9448382808fps with 968 triangles
  torus.obj: 60.8843425227fps with 2048 triangles
  uv-sphere.obj: 60.7561880319fps with 960 triangles
 1280x1024 (1310720 pixels)
 cube.obj: 60.25506445fps with 12 triangles
 ico-sphere.obj: 60.1462824865fps with 1280 triangles
 stanford-bunny-subdivision-1.obj: 60.1538762783fps with 29964 triangles
 stanford-bunny.obj: 60.3821496731fps with 5002 triangles
 suzanne-subdivision-1.obj: 60.1402530504fps with 3936 triangles
 suzanne-subdivision-2.obj: 60.107272135fps with 15744 triangles
 suzanne.obj: 60.1381264119fps with 968 triangles
 torus.obj: 60.5458025167fps with 2048 triangles
 uv-sphere.obj: 60.3641613679fps with 960 triangles
1366x768 (1049088 pixels)
 cube.obj: 60.8836822353fps with 12 triangles
 ico-sphere.obj: 63.6439261931fps with 1280 triangles
 stanford-bunny-subdivision-1.obj: 61.00663517fps with 29964 triangles
 stanford-bunny.obj: 62.644266982fps with 5002 triangles
 suzanne-subdivision-1.obj: 60.613233723fps with 3936 triangles
 suzanne-subdivision-2.obj: 60.9241457455fps with 15744 triangles
 suzanne.obj: 64.2630046024fps with 968 triangles
 torus.obj: 60.7924337015fps with 2048 triangles
 uv-sphere.obj: 64.2945779096fps with 960 triangles
1920x1080 (2073600 pixels)
 cube.obj: 60.0757231929fps with 12 triangles
 ico-sphere.obj: 60.0830990043fps with 1280 triangles
 stanford-bunny-subdivision-1.obj: 60.0781351727fps with 29964 triangles
 stanford-bunny.obj: 60.0976394584fps with 5002 triangles
 suzanne-subdivision-1.obj: 60.1967736842fps with 3936 triangles
 suzanne-subdivision-2.obj: 60.112569145fps with 15744 triangles
 suzanne.obj: 60.2498912462fps with 968 triangles
 torus.obj: 60.220479065fps with 2048 triangles
 uv-sphere.obj: 60.2342107045fps with 960 triangles
```