Contents

Introduction

This report documents the development of the 'Trevi' ray-tracing renderer, and the algorithms and techniques used therein. Named for the Trevi Fountain in Rome and meaning 'three streets', Trevi uses three key techniques to produce high-quality digitial images: basic Whitted raytracing, raytraced ambient occlusion and raytraced diffuse interreflection.

Trevi was developed in two stages: first the basic Whitted component was completed, followed by design and implimentation of the more advanced features. The structure of this report follows this development cycle, and begins by analysing the mathematical foundations of raytracing before moving through the process of implimenting and extending these techniques to produce the completed renderer.

Ray Tracing

Ray tracing is one of two techniques commonly used to generate, or 'render', digital images of three-dimensional objects existing in a defined space, the other being matrix-projected rasterisation. The two approaches differ at a fundamental level; raytracing essentially looks out into the scene, while matrix rasterisation projects the contents of the scene onto a 2D surface. Although computationally more expensive, and thus restricted to 'offline' or non-realtime applications, the raytracing approach is able to produce higher quality, more accurate results as it is based on a model of the behaviour of light. Perspective, reflection, refraction, shadowing, depth of field and many other visual qualitys arise naturally from the raytracing process, while each requires an additional, algorithmically distinct set of calculations when using matrix projection.

The basic raytracing approach, developed by Turner Whitted in the late 1970s and early 1980s, involves firing rays from an eye point into a scene and tracking their interactions with the objects contained by the scene. When an object is encountered, the point of intersection is noted, and depending on the surface of the material a reflection or

refraction ray may be generated and fired into the scene. In this way, the raytracer builds the path that light would take to enter the eye along the initially projected ray, and once each intersection point has been coloured by a lighting model its overall contribution to this path of light can be calculated. By projecting these initial rays through the elements of a raster grid, an image of the scene in 2D can be generated.

Because only the points relevant to each path of light are calculated, raytracing produces very little unused data, except arguably in the case when a lit point will have a visually negligable influence on the final colour of its parent ray projection. However, the process of tracing these rays through the scene is very expensive compared to the cost of projecting the whole scene onto the view plane, as the number of rays calculated for each pixel can climb very steeply when every lighting calculation also uses raytracing to calculate shadows and every intersection can potentially spawn two further rays to trace. Combined with multisampling or supersampling techniques, every pixel of the final image may potentially require thousands of calculations, and thus realtime applications such as videogames are unable to make use of many of the techniques available to raytracing. However, for non-realtime image generation, raytracing forms the foundation of every high-quality rendering solution due to the huge potential it provides in the modelling of the interactions between light and object surfaces.

Algorithms I

A ray-tracing renderer is based upon two main functionalities: the ability to project rays into a scene, and the ability to trace their progress throughout the scene. In order to generate a pixel image at the end of the process, rays must be fired through a raster grid matching the resolution of the final image.

Viewing System

A simple viewing system can be constructed from two components: an origin point, indicating the eye or camera lens position, and a 2D plane of fixed aspect ratio through which to fire rays into the scene. The dimensions of this plane should be proportional to the dimensions of the resultant image, and its location and orientation in relation to the origin point will determine the view of the scene which will be generated in the image.

This plane can be mathematically defined using three vectors:

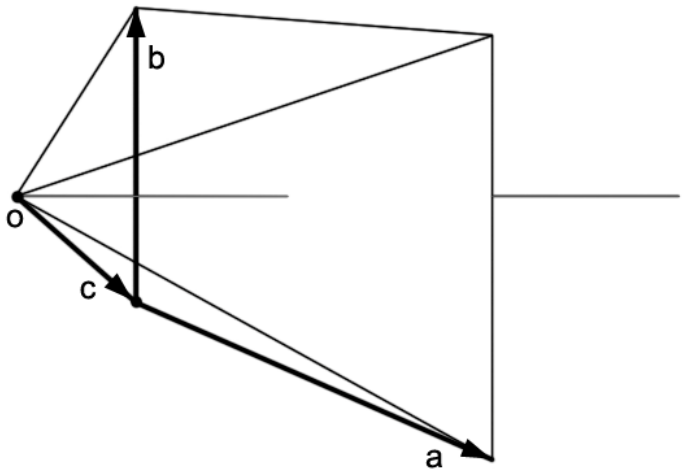


Fig. 1. Defining the viewing plane using three vectors

Vectors **a** and **b** form the x and y axis of the image, while vector **c** locates the plane in relation to the ray origin. Thus, a direction vector through any pixel can be generated by moving proportionaly along the **a** and **b** vectors, as so:

$$\mathrm{Rd}(i,j) = \mathbf{c} + \mathbf{a}*(i / x) + \mathbf{b}*(j/y)$$

where *i* and *j* are pixel locations on an image with dimensions *x* by *y*. Taking the camera origin as the ray start point **o**, we can then define the ray as:

$$\mathrm{R}(i,j) = \mathbf{o} + t*\mathrm{Rd}(i,j)$$
$$= \mathbf{o} + t*(\mathbf{c} + \mathbf{a}*(i / x) + \mathbf{b}*(j/y))$$

for $0 \leq t < \infty$

In practical terms, it is much more useful to define a camera as a position and direction (or position and interest) than using an abitrary plane. To achieve this, it is necessary to be able to calculate suitable values for vectors **a**, **b** and **c** from the direction vector, essentially building the plane mathematically. First, the coordinate axes of the camera, vectors **u, v** and **w** are calculated using the vector cross product:

$$\mathbf{w} = -\mathbf{d}$$
$$\mathbf{u} = \mathbf{w} \times \mathbf{U}$$
$$\mathbf{v} = \mathbf{u} \times \mathbf{w}$$

where **d** is the unit direction vector of the camera and **U** is the world up vector (typically <0,1,0>). This creates a coordinate space where **u** points from left to right horizontally, **v** points from bottom to top vertically and **w** extends negatively into the scene, which is typical for camera coordinate spaces.

In this system, the vectors **u** and **v** form the sides of the viewing rectangle. As the size of this rectangle is abitrary, the horizontal vector **a** is assigned a length of 1, equating it to the unit axis vector **u**. The vertical vector **b** thus becomes the unit axis vector **v** multiplied by the aspect ratio of the final image, as so:

$$\mathbf{a} = \mathbf{u}$$

$$\mathbf{b} = \mathbf{v} * y / x$$

Calculating vector **c**, and thus placing the viewing rectangle, requires an additional piece of information. As the distance *s* between the camera origin and the view plane will control the angle of the view fustrum generated by the camera (see fig. 2), it can be related to the field of view property of the camera.

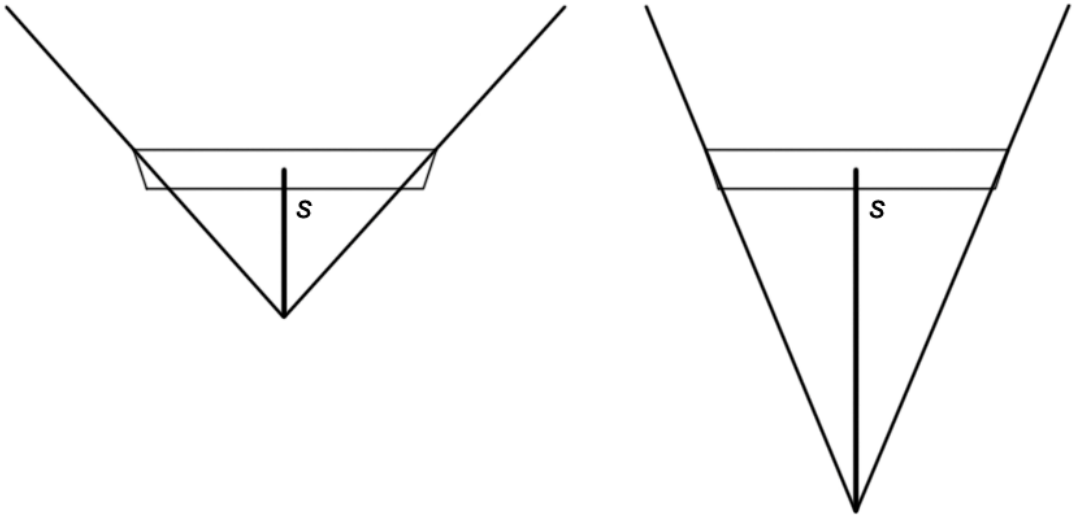
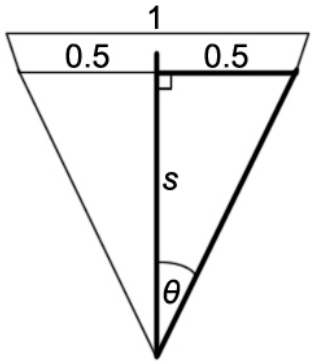


Fig. 2. Adjusting the field of view by varying the distance *s*

Since the viewing angle forms an isocilese triangle, basic trigonometry can calulate the correct distance *s* for any field of view by splitting the fustrum into two right-angled triangles. With the viewing rectangle fixed to a width of one, the *opposite* edge of each right angled triangle will be 0.5 units long, and the angle θ will be half the field of view.

Thus, *s* can be calculated using the tan of θ, as so:



$$\tan(\theta) = 0.5 / s$$

$$s = 0.5 / \tan(\theta)$$

$$s = 0.5 / \tan(fov / 2)$$

Fig. 3. Using the Field of View angle to calculate the distance *s*

From this value, the position of **c** can be calculated by considering the plane's placement in the camera's coordinate axes. As it should be centered along the direction vector (which in camera space is represented by **-w**), the bottom-left corner should be offset from the origin by half its width and half its height in the plane **w** = 0, and then moved *s* units along the negative **w** axis to enforce the field of view. Thus, the vector **c** is calculated as:

$$\mathbf{c} = -\mathbf{a} * 0.5 - \mathbf{b} * 0.5 - \mathbf{w} * s$$

Using these calculated values for **a**, **b** and **c** it is now possible to project a ray into the scene through any pixel location using a camera defined by a position, direction and field of view.

Ray Intersection

Once a ray has been projected into the scene, it will either continue indefinitely, or intersect with one or more of the objects present in the scene. In order to find these intersection points, it is necessary to test the ray against every primitive in the scene. The most common primitive used in computer graphics is the triangle, which is used because it bounds a flat two-dimensional space with the smallest possible number of edges. From a combination of these flat spaces, any more complex surface structure can be constructed.

Other primitive types can be defined and intersected, and here raytracing has an inherent advantage in that as long as a surface can be defined mathematically, a ray can be tested against it. Thus, spheres, cubes, planes and more complex surfaces generated from NURBS or other spline calculations can all be cheaply implemented in a raytracing environment, as they do not have to be tessellated to polygonal form before rendering. However, the triangle is still the core primitive in any rendering program, and it is ray vs triangle intersection which will be considered here.

Generally, when testing for an collision between a ray and a triangle, the ray is first intersected with the plane in which the triangle lies, and then the collision is resolved in two-dimensional planar space. This can be done in several ways, including simply by projecting the triangle onto one of the coordinate planes of world space as dictated by the minor axis of the triangle, but the method described in *Realistic Ray Tracing* by Peter Shirley and R. Keith Morley, and the method used in the Trevi raytracer, uses *barycentric* coordinates. By taking any three points on a plane, the position of a fourth arbitrary point can be determined as the weighted sum of the three coordinate positions, as so:

$$\mathbf{p}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$

If this point lies on the same plane, the sum of the weights will be 1:

$$\alpha + \beta + \gamma = 1$$

For the point **p** to lie within the triangle defined by **a**, **b** and **c**, each of the weights α, β and γ must individually lie between 0 and 1; a higher or lower value will place **p** outside the triangle edges. Combining the above equations gives the equation:

$$\mathbf{p}(\beta,\gamma) = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$$

The point of intersection between this plane and a ray **R** = **o** + *t***d** is thus given by:

$$\mathbf{o} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$$

with the constraints

$$\beta + \gamma < 1$$
$$0 < \beta$$
$$0 < \gamma$$

If the point satisfies these constraints, it intersects the plane within the bounds of the triangle.

The solution to this equation presented by Shirley and Morley uses *Cramer's rule* to solve a 3 x 3 linear matrix equation constructed from the vector form noted above. More details on this solution, which is used in the Trevi raytracer, can be found in their book titled *Realistic Ray Tracing*.

Basic Raytracing

With the ability to project rays into the scene and trace their interactions, it remains to combine this into a complete rendering application. While different applications will likely use vastly different data structures to hold and manage their scene and image information, each will be built on the same basic method of constructing an image:

```
For every pixel in the image
    Project a ray into the scene through the pixel
    Find all ray-object intersections
    If any intersections are found
        Select the closest intersection
        Perform a lighting calculation for this point in the scene
        Set the pixel with the calculated light intensity
```

It is important to note that, being of infinite length, a ray may well intersect with multiple primitives in the scene. However, as the ray models the path of a beam of light, only the first surface it encounters is of interest, as at this point the light interact with the surface and thus be prevented from continuing on its path. To find this point, the ray equation $\mathbf{R} = \mathbf{o} + t\mathbf{d}$ is considered; since $\mathbf{d}$ is a unit direction vector, the value of the parameter t is the distance along the ray that the intersection occurs. Fortunately this value is calculated as part of the intersection test, and thus can easily be saved for later reference. Selecting the lowest positive value of t (since negative values indicate an intersection behind the start point of the ray) identifies the closest intersection, and this is the one which is processed.

Although this basic algorithm provides a complete image of the scene, the real power of raytracing is in modelling the transit of light from point to point in the scene. Thus, raytracing renders make use of recursion to gather lighting inputs from around the scene by constructing a *ray data tree* containing every point in the scene which will contribute to the point being lit. The ray data tree is a binary tree containing two fork

directions: reflection and refraction. By continuing to generate rays on the leaves of this tree until either a non-reflective/refractive surface is reached or a pre-defined recursion limit is reached, the renderer can identify every contributing point, and perform lighting calculations for each. Thus, the basic algorithm can be expanded as so:

```
For every pixel in the image
    Create an empty ray data tree to a specified size
    Generate the initial ray through the pixel and store it in the tree
    While the tree contains an unprocessed ray
        Find all ray-object intersections
        If any intersections are found
            Select the closest intersection
            Is the surface reflective?
                Generate a reflection ray and store it in the tree
            Is the surface translucent?
                Generate a refraction ray and store it in the tree
    For every ray intersection in the tree, starting at the leaves
        Perform a lighting calculation for this point in the scene
        Blend the colour with reflected/refracted colours according to
        surface properties
    Select the final colour at the initial ray intersection
    Set the pixel with the calculated light intensity
    Clean up the ray data tree
```

Intersections can be lit using basic local illumination models, although since the ray data tree implements specular reflection of light a model with a specular term, such as the Phong or Blinn models, should be used. Raytracing lends itself to shadowing very naturally, as all the mechanics to test for shadows being cast are already implemented in the application. A ray is generated from the light source to the target point, and

tested against the objects in the scene. Unlike rays cast to trace visibility through the scene, shadow-cast rays do not need to test against every object; once an interposing primitive has been identified, there is no way for the light to continue on its path and the target can be considered to be shadowed. However, it is important to test that any intersection found does indeed fall between the light and the target, as objects further from the light than the target point cannot cast shadows on it. Likewise it should be checked that the interposing primitive is not in fact the object being shadowed; due to limits on the accuracy of floating point and double length decimal values, it is sometimes possible for a surface to incorrectly shadow itself.

Implementation I

Implementing the basic Whitted raytracer as described above involves rendering the relevant algorithms into code, and creating an efficient structure to provide a data flow to and from these algorithms. Ideally such an application should be data driven, capable of loading configuration options and scene data from an external file to maximise its flexibility.

Renderer Design

The Trevi renderer is built upon a modification of design techniques developed for a matrix projection renderer, primarily in terms of scene storage. Objects are stored as two distinct components: models and meshes. Meshes contain raw object data; currently only explicit vertex/index data for generation of triangle primitives is supported, but this could easily be extended to include both procedural primitive generation and implicit surfaces such as NURBS or free-points. Models contain mesh instantiation data, such as placement transformations and surface properties, and are linked to a specific mesh by reference. In this way, a single mesh definition can be reused by any number of models without the need for duplication of data.

Trevi models also include surface and material data, which includes the standard ambient/diffuse/specular terms as well as an option to switch between smooth normal interpolation and flat triangles. Surfaces can be assigned to subgroups of triangles by ranges or individual IDs, allowing one mesh to be drawn with multiple surface properties representing different real-world materials. Support is also in-place to allow for texturing of these surfaces, although this is not implemented in Trevi at this time.

The scene data supports any number of cameras and lights within the scene, all loaded from an XML format designed specifically for Trevi. AISXML directly matches the internal data structures of Trevi, allowing for quick loading and easy scene design. The format

also includes renderer configuration settings such as output and supersampling resolutions, which are stored on a per-camera basis to allow a scene to be easily rendered multiple times from a single file.

In order to speed up the rendering process, a large number of static properties are pre-calculated before rendering begins. This includes the edges and normals of every triangle as well as the coordinate axes of the camera. This trades storage for speed, as it requires each model to cache its indexed vertex arrays in a unique triangle array containing pre-calculated values, negating the runtime benefit of the separate model/mesh structure. However, this structure is not rendered redundant, as it simplifies the scene files and still allows for non-explicit primitives to be used.

As well as containing cached values to speed up rendering, the triangle data structure contains the implementation of the ray-triangle intersection algorithm described previously. Although this algorithm could be placed elsewhere in the application, since triangles are the only primitive supported by Trevi, including it in the triangle class opens the way for non-triangle primitives and procedural surfaces, which could be implemented through a parent 'shape' class with numerous derivatives without affecting the structure of the rest of the application. The same approach can be seen in the light data type, which contains subtypes for point and spot lights which are indistinct to the lighting model.

Implementation Overview

Trevi was initially developed as a number of separate components comprising the key components: the viewing system, intersection module and scene management tools. This allowed each component to be tested in isolation before being combined in the final application, making bug-fixing easier.

It was originally intended to use a different method to the one presented when developing the intersection module. As mentioned in the algorithm section, ray-triangle intersections can be detected by projecting the 3D plane containing the triangle and an intersection with this plane onto one of the coordinate axes of the world space. Placing the intersection point in relation to the triangle is then calculated barycentrically, similar to performing the calculation in three dimensions, but with a simplified calculation due to less component values and the elimination of the use of matrices. This approach trades accuracy for speed, attempting to minimise the inherent error by selecting a projection axis which will result in the largest two-dimensional area for the projected triangle. Unfortunately, although this algorithm was successfully implemented in a shadowing context, adapting it for use with a more general raytracing solution proved problematic, and ultimately it was dropped in favour of the method described in *Realistic Ray Tracing* in order to allow time to develop more advanced features using the raytracing core.

The scene management component underwent several major iterations before settling on the code used in the final version of Trevi. In particular, different methods of storing surface and material properties were explored, including storing a single scene-wide list of materials to allow multiple models to use the same material settings. However, this led to problems with maintaining surface assignments on specified triangles within a model, and so was replaced with the current method in which surfaces are stored as a property of the model itself.

Maintaining these surface assignments remained a persistent issue through development of the scene management module, as assignments are performed through direct pointers to the surface to decrease lookup time, since the triangle->surface link is accessed heavily by the rendering routine. Issues arose when adding new surface properties to a model which already contained surface to triangle assignments, as these pointers would be invalidated by the relocation of the surface array when its size was dynamically increased. This issue manifests itself as garbage data appearing on all but

the most recently added surface, often causing a primitive to appear pure white in the rendered image. Once identified, the problem was solved by storing an offset value between the old and new memory addresses of the surface array, which could then be recursively applied to the saved pointers. One downside of this approach is that the model class is not 64-bit compatible, as the pointer length on a 64bit system is over that of the integer used to store the pointer offset. However, this could be overcome should 64bit compatibility be required by updating the code to use a larger field to store the value. An alternative approach would have been to assign surfaces by ID during scene construction and cache a pointer just before the rendering process begins.

Another area which had to be carefully designed was that of reflection/refraction in the ray data tree. Since a ray should refract upon both entering and leaving a medium, a method of tracking what substance the ray was passing through was required. Initially this was tested as a flag to indicate if the ray was in air, which would be used in the intersection testing to determine whether to flip a triangle before calculation, since of course a polygon hull is one-sided. However, this proved complicated to track through the numerous function calls involved in the rendering process, and for simplicities sake it was decided to instead require the scene file to take care of inner-surface refraction by placing an inverted mesh in the same space as that used for external surfaces. Since this mesh would not appear to a ray entering the object it does not affect anything but the ray leaving the object, and also allows for fine-tuning of the refracted image as it has its own unique refractive index. To support this mechanism, the AISXML format was extended with an 'invert' flag on a loaded mesh, which performs inversion of both the vertex normals and polygon winding in order to provide a correctly inverted mesh.

Blending of reflected/refracted colours is performed by a simple linear interpolation between the local illumination value and the reflected/refracted illumination value according to the reflectivity and opacity surface properties respectively. Initially the order in which this was done was arbitrarily decided to be reflection followed by refraction; however, once scene files were being rendered it was decided that reflected

light should not be diminished by the translucency of the reflecting surface, and so the blending was reordered to allow reflection to occur at full intensity.

Testing

By developing the renderer in three distinct components, it was easy to test the correct operation of each subsystem before the full program was assembled. As each component came together, it could then be used to assist in the testing of the next section, starting with the ray intersection system.

In order to test correct intersection detection, known rays were fired at known object locations in a simple console environment, outputting a boolean result to the command line. Initially triangles were places within the coordinate planes and tested against perpendicular rays; for example, testing a triangle <0,0,0><0,0,1><1,0,1> against a ray with direction <0,-1,0> from positions directly above the triangle centre, edges and corners and at locations outside the triangle (see fig. 4)

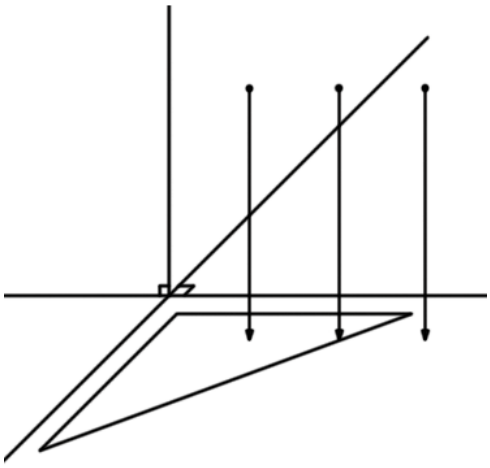


Fig. 4. Testing rays against a known triangle

Once single-direction intersections had been tested, a number of three-dimensional triangle/ray pairings were constructed on paper using results from other rendering applications, and tested mechanically in the code, again simply outputting a true or false

statement for each test case. Finally, an array of predefined rays was fired at a number of hard-coded meshes such as triangles and cubes and output as a black/white image indicating the presence of an intersection, in order to test the code's ability to identify object silhouettes.

This final test was adapted for use with the viewing system tests, where the predefined array of rays was replaced with a batch of calls to the camera to generate the same set of rays on the fly. Again, only silhouetting was tested as at this stage no scene data or lighting was implemented. The camera was tested at a number of resolutions and aspect ratios to observe how the algorithm dealt with such changes, and the maths was tweaked on several occasions to ensure uniform, predictable behaviour.

Finally the scene management component was added to the application, and could immediately be tested for correct rendering operation by drawing complete scenes thanks to the certainty that the camera and intersection components were behaving as expected.

Algorithms II

Once the basic Whitted specification had been completed, Trevi could be extended with advanced techniques to improve the quality and realism of rendered images. Primarily, it was intended to address the major shortfall of raytraced images, diffuse inter-reflection, and this is discussed later in this section. First, several simple techniques inspired by Shirley and Morley's suggestions were implemented.

Depth of Field

The viewing system outlined previously is modelled on a pinhole camera, which in practical terms means every ray begins at the same origin point. This results in an image which retains perfect focus at every distance, which is not an accurate depiction of vision. As the human eye views the world through a lens, it inherently has a focal point at which perfect clarity is seen, which falls off as distance from this point increased. This effect is called depth of field, and can be easily simulated in a raytracing environment.

Since the camera in the previous viewing system generates all rays from the same origin, the first step is to randomise the starting point of each ray. This in turn affects the angle at which the ray travels and thus the path it takes through the scene, creating noise. By performing this process several times for each pixel, for example by supersampling the image as Trevi does, the resultant colour will be an average of several lit points, which depending on proximity leads to a greater or lesser amount of blur.

Randomising the origin of a ray, or 'jittering', can be done quite simply by specifying an area to jitter over and then randomly moving the origin along the **u** and **v** axes of the camera system – effectively translating the ray origin parallel to the viewing plane. Since this is being done to simulate the lens on a camera, this area can be called the 'aperture' and can be specified by a simple length. Depending on the effect desired, this length could be the radius of a circular lens, or else just the dimensions of a square. Since

randomising position over a circle requires trigonometric calculations, Trevi uses a square lens to increase render performance.

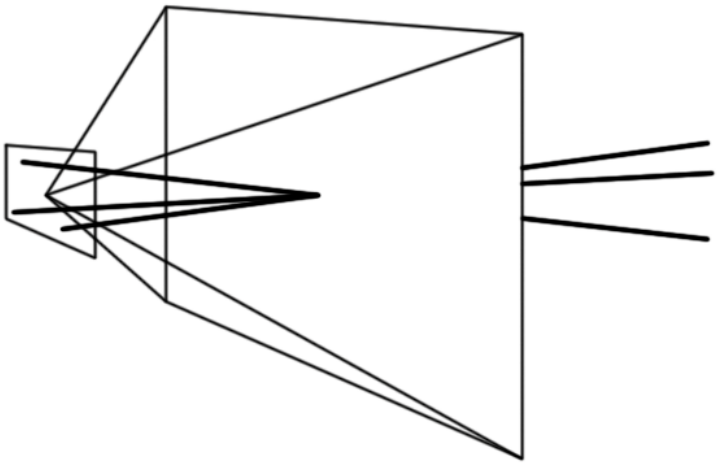


Fig. 5. Jittering the ray origin in an aperture plane naturally creates a focal point

However, merely jittering the ray origin in a 2d plane will produce a depth of field effect with a fixed focal point – specifically the distance *s* as determined by the field of view calculation, since the ray is projected straight through this pixel. To adjust the focal length of the camera, the convergence point of the jittered rays must be variable.

This can be achieved by first performing the standard un-jittered ray calculation, to obtain a perfectly focussed ray entering the scene. A focal point can then be selected at any point along the length of this ray and used with origin jittering to produce a sample of rays which will converge at any distance from the eye, giving a variable focal point.

This technique can also be applied to shadow-casting ray intersection tests, where jittering the position of a point light source will produce a realistically soft edge to shadows when a high sample rate is used. This simulates the fact that physical lightbulbs emit light from a volume of space rather than a single infinitesimal point.

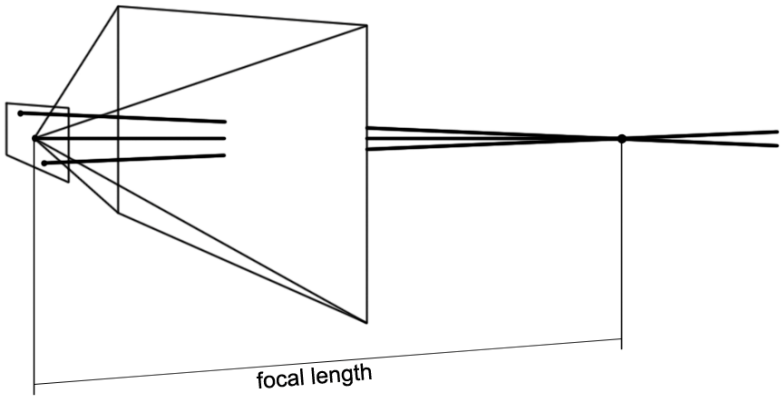


Fig. 6. Projecting jittered rays to an arbitrary focal length

When applying jittering to shadow casting, it is not necessary to calculate a focal point for the ray, as it will still be traced towards the intersection point being lit. However, it is necessary to calculate coordinate axes for the light to allow the origin of the ray to be jittered perpendicular to the original direction. This is substantially cheaper than jittering the light across a hemispherical area, which although closer to the physical property being modelled is unlikely to be visually distinguishable once sampling has taken place. Calculating these axes is done in the same way as for the camera in the original viewing system, using the cross product with a defined up vector, and uses the direction from the light to the intersection point in place of the camera's specified direction vector.

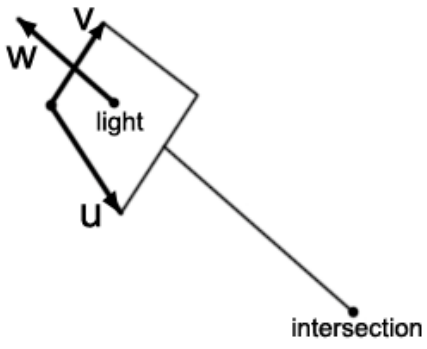
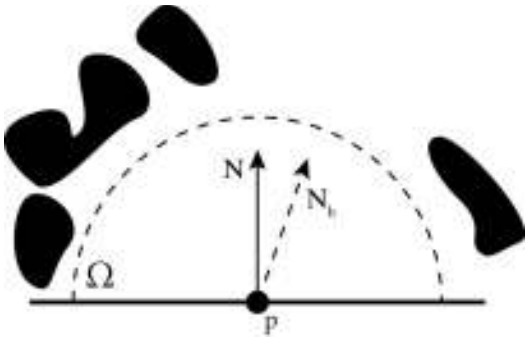


Fig. 7. Coordinate axes can also be generated to jitter light sources

Ambient Occlusion

Ambient occlusion is a pseudo-global illumination technique used to cheaply model some of the behaviours of a true global illumination model, specifically the soft shadows created when objects are situated in close proximity to each other. This occurs due to a lesser amount of ambient light being able to reach these enclosed spaces, and can very easily give a real view of solidity to a computerised image, which may otherwise look very artificial despite sophisticated lighting simulation.



$$A_p = \frac{1}{\pi} \int_\Omega V_{p,\omega} (N \cdot \omega) \, d\omega$$

Fig. 8. Ambient occlusion approximates attenuation of light due to occlusion
(Image from http://en.wikipedia.org/wiki/Ambient_occlusion)

The ambient occlusion equation integrates a visibility function *V* over the hemisphere Ω to determine the proximity of any objects which may occlude incoming light sources, and thus calculate a factor by which to darken the surface, indicating a reduction in ambient light in that area.

By randomly sampling the hemisphere, a technique known as Monte-Carlo integration, an impression of the occlusion factor for a point can be gained, which increases in accuracy the more samples are taken. These random samples can be obtained by rotating the normal vector by a random angle in two axes – first around any axis which lies in the plane of the intersection point, to adjust the elevation, and then around the original normal itself to adjust direction.

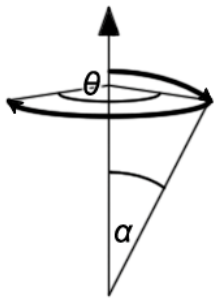


Fig. 9. Rotating a normal vector to generate a random direction for sampling

This new ray is then traced across the scene to identify possible occluding intersections. At this point, the standard ambient occlusion equation would use a visibility function returning 0 if the ray is occluded and 1 otherwise, and attenuate this value by the cosine of the angle between the sample ray and the intersection normal, calculated using the dot product. This means glancing rays have very little impact on occlusion, while those directly above the surface will darken it significantly. However, this equation does not take into account the distance between the surface and occluder, relying on the fact that a closer object will occlude more of the sample rays.

Since Trevi was designed with a Cornell Box environment in mind, and this box forms a completely enclosed space, it was thus certain than every sampling ray would meet an occluder within the scene. Logically this causes issues with the standard ambient occlusion technique, as the occlusion factor will always equal zero. Instead, Trevi attenuates the influence of an occluder by its distance from the point of intersection by linearly interpolating across a fixed distance. Thus, Trevi's occlusion visibility function returns **1** when no intersection is found and **$1 - (s_i / s_{max})$** when intersection occurs, where s_i is the distance between the occluded point and the occluding object and s_{max} is a pre-defined maximum distance. Since a scene could potentially be any dimension, the value of s_{max} is pre-calculated from its diagonal dimensions, from corner to corner of a cube encompassing every object in the scene. In this way, ambient occlusion can always be calculated using reasonable values regardless of scene specifics. Low resolution testing suggested that **diagonal / 2** provides a suitable value for s_{max} in a roughly cubic

scene like the Cornell Box, providing a decent level occlusion contrast without over-darkening the scene. Of course, the quality of the resultant effect depends entirely on the number of samples used to calculate the occlusion factor, as the technique is highly susceptible to noise.

Diffuse Raytracing

As noted above, the major flaw with raytraced rendering is that it is unable to simulate diffuse inter-reflection, where colour from an object will 'bleed' into surfaces around it. In the real world, this effect occurs because a diffuse surface absorbs all wavelengths of light except for those that make up its surface colour, and scatter these remaining wavelengths evenly from the surface. This is what gives objects visible colour, but it also results in small intensities of coloured light radiating from any surface, which may in turn be reflected in those around them as if directly lit.

Since diffuse light is radiated evenly over the hemisphere around any point, it is not modelled by standard raytracing, which reflects in a purely specular manner. Many techniques have been developed to simulate diffuse inter-reflection in other ways; radiosity is one such purely-diffuse rendering technique, which is often combined with a further specular pass to produce a more complete image. However, it is possible to model these interactions using the Monte-Carlo sampling technique described above.

Using the same method of randomising the normal direction, Trevi projects a number of sampling rays into the scene and performs local illumination calculations for any discovered intersections. By treating each of these intersections as an additional light source and performing a lighting model calculation at the original point, it is possible to calculate a sample contribution of diffuse light gathered from around the scene, particularly when each sampled point is recursively diffuse-traced. As with ambient occlusion, the contribution of each such 'light source' is attenuated with distance to prevent a massive build-up of light apparently being emitted by every diffuse surface.

While it suffers from the same sampling-rate requirements of raytraced ambient occlusion, such diffuse raytracing is able to produce a reasonable approximation to the diffuse bleed effect at a much lower cost than a fully global radiosity solution, as it only calculates diffuse reflection for visible points in the scene.

Implementation II

Unlike the initial implementation steps, developing the advanced features described above produced instant results in the visible scene. Once the basic framework for each effect was in place, much of the development time moved over to tweaking the equations to produce more aesthetically pleasing results.

Design Modifications

The most important consideration when implementing ambient occlusion and diffuse raytracing was where to insert the calculations in the rendering process. If included in the standard ray data tree loop then the set of initial calculations could be saved, as the module could just divert the first intersection result for each pixel at a convenient point in the loop. However, this makes the components reliant upon each other to perform, breaking down some of the modular nature of the Trevi design; performing and combining multiple passes to produce the finished image would be a much more elegant solution. After some tests this second approach was decided upon, since it would make the software more flexible in the tasks it could perform. To optimise the render time as much as possible, rather than recalculating the first set of intersections used by the diffuse and occlusion modules, these would be saved to a buffer and re-used as required by the other components.

The diffuse raytracing component did require one key modification to the standard rendering loop. Since the diffuse lighting would be calculated separately to the main pass, the Phong lighting model function was split into its diffuse and specular

components, to be used at separate points in the algorithm. The original function was also retained to allow non-raytraced diffuse lighting to be performed without repeat calculations if the diffuse raytracer was turned off for an image.

Depth of field was somewhat simpler. Since Trevi is designed with supersampling in mind, it was not necessary to repeatedly sample the jittered primary rays, nor the light positions for soft shadowing. Although performing this sampling at a code level would allow for smoother transitions at low supersampling rates, it would generate the same number of raytracing tests as would be performed in an image rendered at a higher supersampling value, and which was deemed unnecessary. Thus, depth of field and soft shadowing could be inserted straight into the camera and light classes without touching any other area of the renderer.

Implementation Overview

Being based on fairly simple, mechanical equations, implementing the advanced features described above was a straightforward process, involving more rearranging of existing code to allow the new sections to slot in than overly complex software development once the initial theoretical legwork was done.

In order to prevent having to pass numerous flags and sample counts through a chain of function calls, the control variables for the ambient occlusion and diffuse raytracing modules were added to the camera class alongside the general settings such as image resolution. This allowed the majority of function calls to remain simple, requiring only such arguments as required by the body of the function, as well as adhering to the design laid out during the first implementation phase. A separate configuration class was considered, which could have been either a singleton or a single argument to be passed into all relevant functions, but it was decided to retain the per-camera settings basis to allow for highly configurable rendering options.

One interesting effect occurred while implementing the jittering effect for depth of field and soft shadowing. Both techniques make use of the rand() function, which requires seeding using srand(), and in isolation each worked without issue. However, when both were used all random sampling disappeared from the scene, which instead began to tear in alternate vertical stripes of about 15 pixels. A quick test identified this as resulting from repeated use of the srand() function, which appeared to interfere with the generation of camera rays, and so the seeding was moved to a generic location in the rendering pipeline.

The largest part of this implementation phase was testing the new effects and tweaking the relevant equations to produce more appealing results. This is because Trevi uses non-standard equations to perform these tasks in a manner suited to the scenes to be rendered, and in particular a lot of time was spent trialling different approaches to the ambient occlusion calculation once the samples had been fired.

Testing

Initially, all features were tested on a low-resolution image to allow for speedy rendering times when making minor adjustments to code or equations. Once complete, the renderer was tested with higher quality settings, using resolutions closer to the required 3000x2000 output size coupled with 2 or 3 times supersampling, and here problems arose.

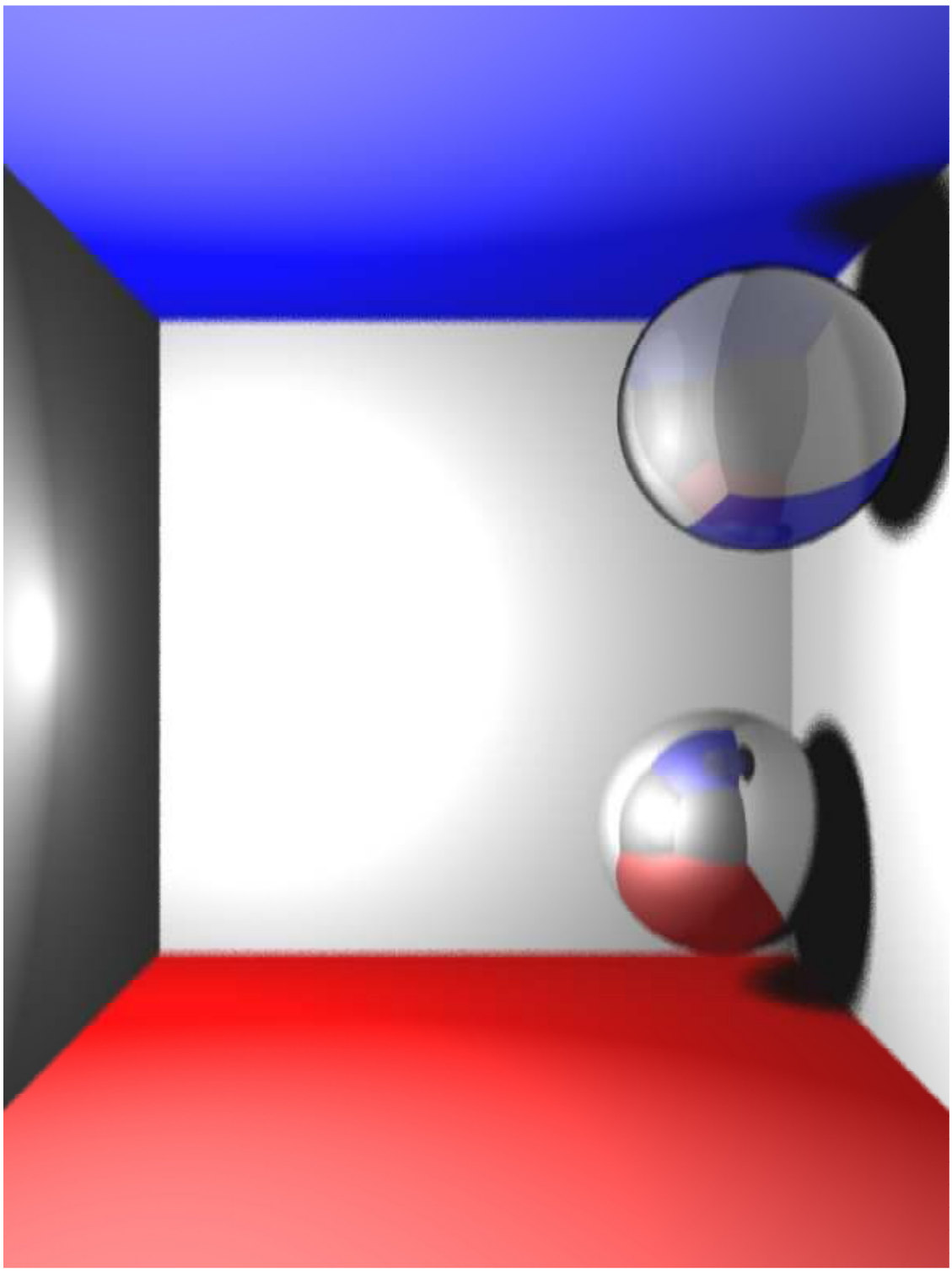
Due to the size of buffers required to operate at such high resolutions, Trevi was using a huge amount of memory to perform its calculations, so much so that it would frequently crash while trying to allocate memory using **new**. In particular, the amount of memory required to store the first intersection of each pixel for an image of several thousand pixels per side became astronomical, with a 3000x2000 image at 2x supersampling requiring just over 1.6gb of ram purely for intersection data. Obviously this is not acceptable memory usage, and the render process had to be drastically reworked,

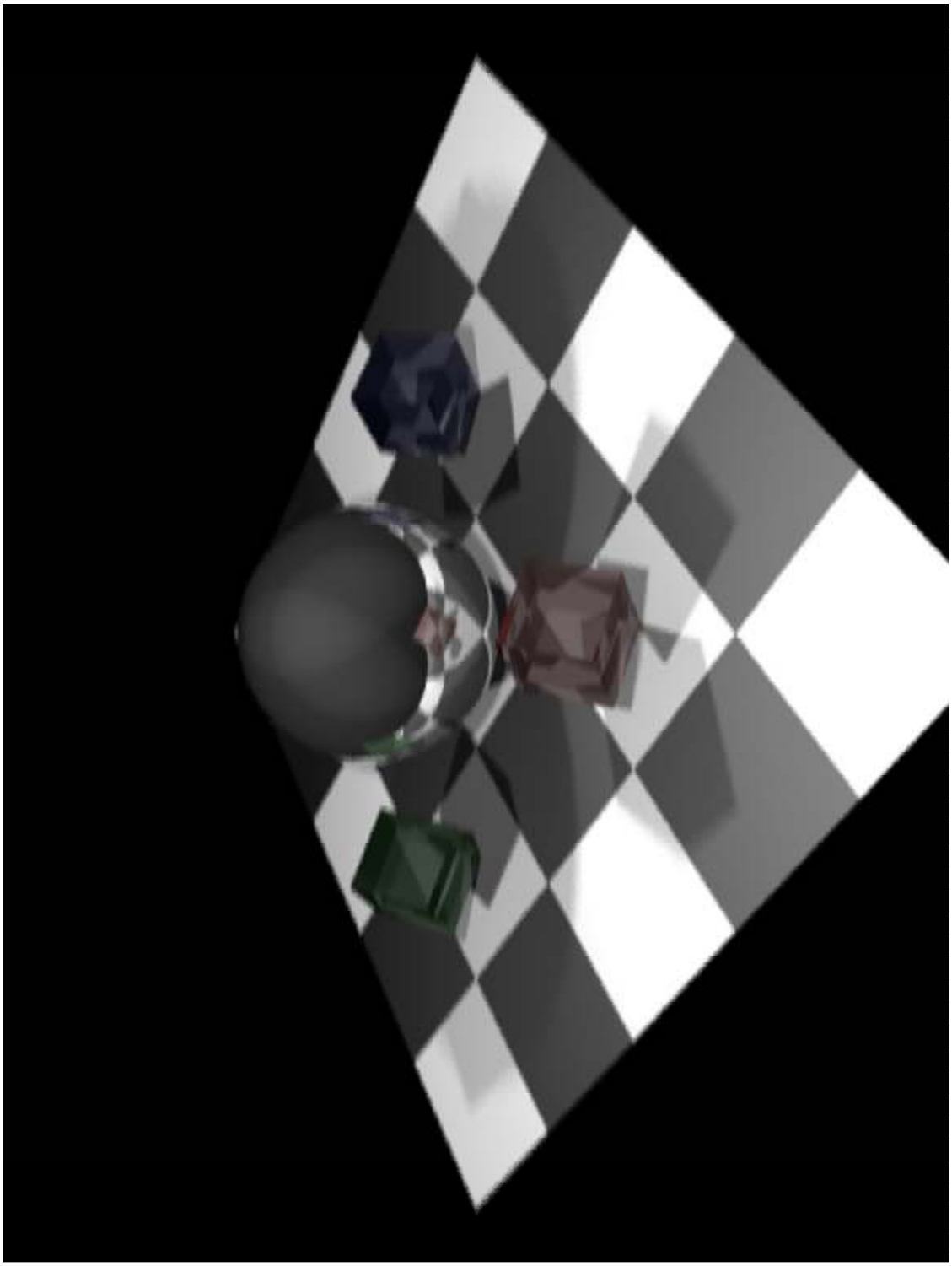
combining all features into a single pass to prevent massive buildup of required storage. Although this is a less elegant solution than the previous pass-based approach, it became necessary when the full memory usage of the application was realised.

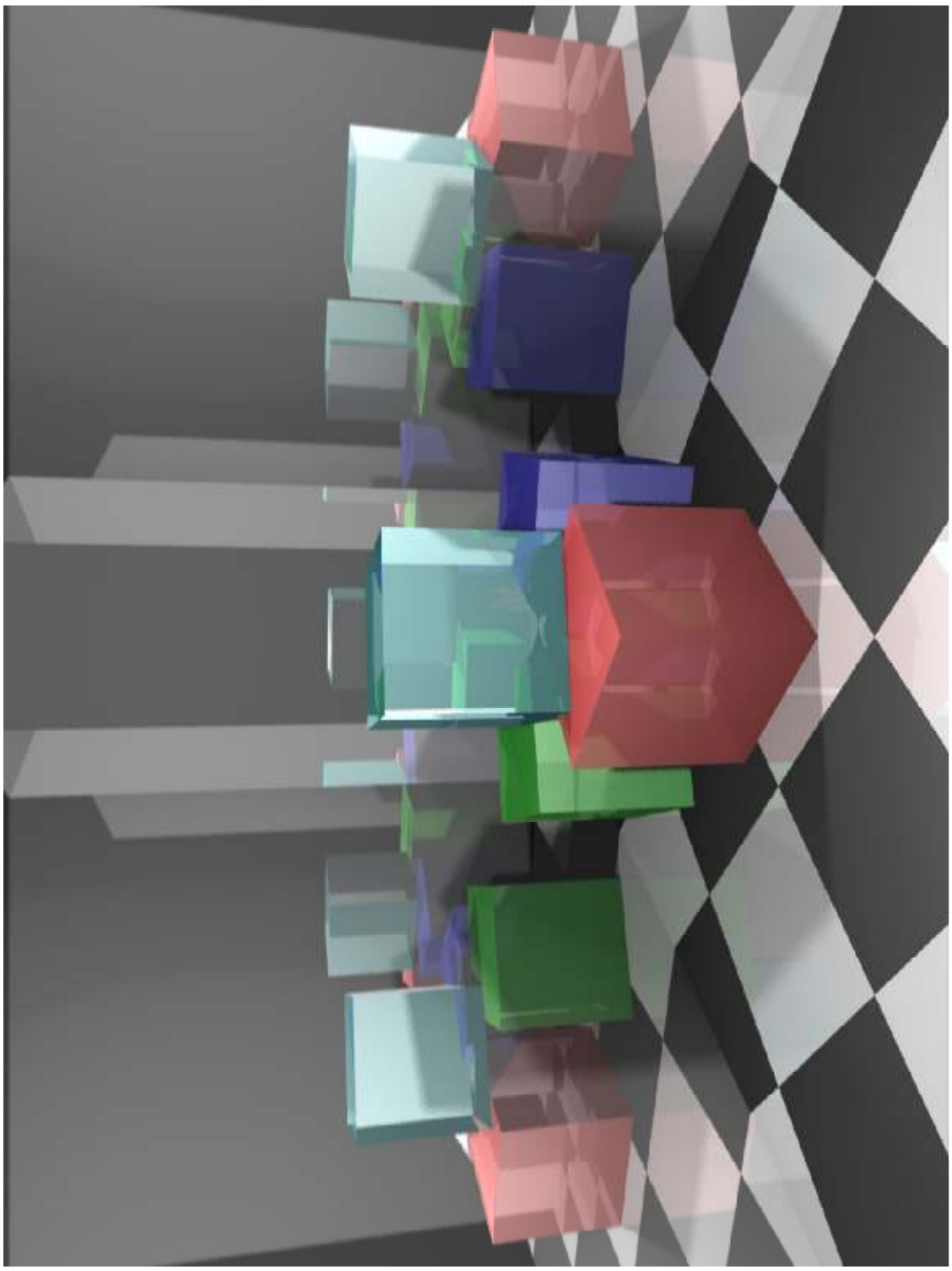
The processing performance of the application was also highlighted while testing its performance at higher resolutions. While supersampling naturally has a large inherent cost, the high sampling requirements of ambient occlusion and diffuse raytracing easily overshadow this, slowing the render process to a crawl when enabled. In particular, combining supersampling with these other techniques proves prohibitively expensive, despite being required for correct interpretation of the sampling results. While as many optimisations as possible were implemented to attempt to combat this, the core raytracing engine still has much room for improvement, and unfortunately the render times for larger images reflect this.

Sample Images

The following sample images were rendered to a resolution of 3000x2250 with 2x multisampling (4 samples per pixel). Due to time constraints it was necessary to render these images without using the ambient occlusion or diffuse raytracing components. All three images plus the scene files used to render them are included on the accompanying CD.







Performance

As noted above, Trevi regrettably has some serious performance issues in its current form. Most problematic of these is that of memory usage; even without caching intersection results, the average footprint of the application while rendering a 3000x2000 scene at 2x supersampling remains around 800-1000mb, enforcing a huge restriction on the quality of image which can be produced.

This issue arises in full from the size of buffered data required for rendering such high resolution images with all the advanced features, and addressing it must be the highest priority in order for Trevi to progress as a viable raytracing solution. Analysing the rendering process highlights a very plausible fix for this problem.
The raytracing algorithm as described initially in this report and developed further for Trevi renders each pixel in the image individually and without reference to the rest of the image. Thus, it should be possible to entirely eliminate the need to buffer the entire image at once by writing the resultant pixel data to the output file as it is generated, allowing the renderer to entirely clean up after every batch of calculations. Even allowing for supersampling, it should be possible to render clusters of pixels and write these to disk before moving on. Unfortunately there was not time to make this major change in file handling within the constraints of the project, as higher-resolution testing was only done towards the end of the timespan and thus the problem remained unnoticed for several months. However, correct implementation of such a solution could entirely clear up the memory issues currently associated with Trevi, allowing for potentially any size of image to be rendered with much larger sample rates than are currently possible.

Unfortunately, once this issue is dealt with another will have to be immediately tackled – that of render speed. As noted above, images rendered using ambient occlusion and/or diffuse raytracing require vast amounts of samples to be taken, which means a huge number of rays must be traced against the scene. The complexity of the scene also has a direct impact on the time taken to render, with processing time for even a few

thousand triangles soaring to astronomical levels, as can be seen in the time taken to render the sample images provided with this report.

Clearly the core raytracing algorithm must be looked at, and particularly spatial partitioning techniques will be required to cut the time taken to trace a ray against the scene. Since testing a ray against a triangle is so computationally expensive, every possible step must be taken to reduce the number of times this must be performed, even while simultaneously upping the sample rates to increase image quality, and the best way to achieve this is to eliminate the bulk of triangles from consideration before the need to test them directly arises. Spatial partitioning techniques such as BSP/quadtree/octree are ideally suited for this task, and must take priority second only to resolving the memory management issues in the continued development of Trevi.

Improvements

While it has some serious performance issues in its current form, Trevi remains a robust and flexibly designed renderer, with room for expansion into a number of areas. Most obvious of these would be the inclusion of implicit surfaces and procedurally generated primitives, which would go a long way towards reducing the triangle counts of more complicated scenes while also allowing for much more flexibility in scene specification.

Since the triangle class already contains the code for testing a ray against the triangle primitive, it would be easy to expand into a parent ‘shape’ class and produce a complete library of possible object types. Consideration would have to be given regarding the assignment of surface properties to subsections of these procedural shapes, which could be done using ranges of UV coordinates or of any variables in the defining equation.

Trevi would also benefit from a wider range of advanced simulation techniques, to make it more versatile at rendering a wider range of scenes. It would be particularly suited to the implementation of beam tracing thanks to its raytracing heritage and early steps

into area lighting, and the inclusion of lighting through refractive surfaces would round off its feature set to provide a viable solution for every form of light transport.

In the further future, it would be very interesting to expand Trevi to include a true global illumination model such as photon mapping, not least to compare the quality of the images that could be produced with those it currently generates through simpler approximations to the behaviour of light.

Conclusions

Although currently experiencing some teething troubles in the area of runtime performance, the Trevi raytracer nonetheless carries a robust feature list and is eminently capable of producing high quality images when given time to work. By including ambient occlusion and diffuse raytracing, Trevi steps up from the basic Whitted model renderer to provide a more complete rendering solution with a lot of scope for customisation and further expansion.

The development of this application, a few minor hitches aside, has been a remarkably smooth process and the end result bears this through. The largest issue to have arisen is that of Trevi's performance at higher quality settings, which would have become apparent much earlier during development if comprehensive testing had been carried out at both ends of the scale. While unfortunate, this shortcoming has taught a valuable lesson, and once these problems have been resolved development of this renderer will continue apace.

In the meantime, the quality of image produced by Trevi even at its lower settings is something to be proud of, and the prospect of continuing to expand and improve upon its operation is anticipated to be as interesting and enjoyable an experience as the project has been to date.