# 1 Introduction

Traditionally models are textured by using surface textures which are applied to the outside of the object. Using textures in this way means that if the size of a model or portions of the model removed the texture is no longer valid and will stretch incorrectly.

Solid textures do not have this inherent problem as they are not connected to the object but are their own 3d object. This means that subtracting portions of a model or changing the shape doesn't warp the texture.

By allowing a solid texture to be created as a combination of multiple solid textures as a tree based model it is possible to create an interesting variety of materials.

This dissertation aims to implement a system that is capable of rendering these materials. Allowing them to be reflective, refractive and have changing colours throughout the material. Also by making use of combined solid geometry modelling to make it possible to view the innards of a material created by this method.

The outline of the requirements are found in the requirements specification in section 3. The design of the system may be found in section 4, details of the implementation of the system in section 5. Details regarding the testing of the system may be found in section 6 and conclusions may be found in section 7.

# 2 Literature Review

## 2.1 Introduction

In the world of computer rendering there are many different methods for rending objects within a computer. Such systems use surface models, voxels or particle systems to produce an output. These systems make use of the detail of the models to create the rendered image. It is also possible to detail the objects with the materials and textures that are applied to these instead or merely in addendum to the object.

## 2.2 Combined Solid Geometry Ray Tracing

Combined Solid Geometry, henceforth referred to as CSG, is a method of modelling used that works using boolean operations on to combine mathematical objects together. CSG is performed on objects that are held within a binary tree. The different boolean operations used in CSG are the following.

- Union – Union is where the area of the objects are combined together such that all the area of the objects are represented
- Merge – Merge is similar to union but the area within multiple objects is not made of its component parts. Merge appears to perform the same operation on objects as union but produces a different result when the material of the objects are transparent.
- Subtraction – Where the area of objects on the right branch of the tree are subtracted from the area of objects on the right.
- Intersection – Where the area of objects only exist where area exists in both branches of the tree.
- Symmetric Difference – Where objects only exist where there is one or the other but not both.

Ray tracing as described by siggraph[8] is a “global illumination method”. Which casts rays from the eye into a scene and is tested against the objects in the scene to determine whether it intersects with any object. Ray tracing is a simple method for implementing shadows, reflections and refractions.

Ray tracing is a highly parallel rendering method and thus is very easy render in a multi threaded system.

## 2.3 Introduction to 3D Textures

3d Textures are those that aren’t applied to the surface of a model but one that notionally exists throughout the model, whether this is visible or not. The concept of a “solid texture” [1] was introduced by Ken Perlin. Solid texture works as a "space function" that has 3d coordinates. A solid texture thus exists in 3d space and is independent to the volume to which it is applied. Since the texture is independent and continuous through each dimension the texture can fit any shape it is applied to.

Compared to a standard 2D texture, solid texture has the benefits of not having to custom design each texture per shape it is applied to. Since the texture has an extra dimension the space required to store the texture would be substantially increased. Due to this reason, coupled with the difficulty in manually creating a texture that would tile effectively over every dimension, 3D textures are usually created procedurally.

## 2.4 Procedural Textures

Procedural textures are those that are made by procedure to calculate the colour at a particular location within the texture. The traditional method is to use an image file and perform a lookup on the image when applying it to a surface. Procedural textures have the benefit of being able to work at any texture size and the database size is very small. The negative is that they can take time to calculate the colour compared to an image lookup. Procedural textures initially were initially proposed to work over a two dimensional basis.

Perlin extended procedural textures into three dimensions to create solid textures. In addition to using standard mathematical functions to produce the colours, for example sine waves and polynomials, Perlin introduces a few other functions for use in creation of solid textures. These are as follows.

## 2.5 Noise

Noise takes as input a point in 3-Space and generates a value in the range [-1,1] by the combination of pseudo random values corresponding to integer values in the three cardinal directions. The method to generate noise was improved by Perlin in 2002[2] to correct deficiencies in the original noise method. This was done by using a higher ordered quadratic in the combination of the three values of the input such that first and second order differentials had zeros at the same points. The integer lattice was increased to take the corners around the point rather than the faces thus removing the directional bias inherent in the original.

Noise can be used in of itself to produce a solid texture colouring or used in addendum to other functions to add to the texture produced.

### 2.5.1 Dnoise

Dnoise is the "vector valued differential of the noise signal". Dnoise will give the instant change in noise within the solid texture. This can thus be used to perturb the normal to the surface of the object the solid texture is being applied to. As described by Blinn[3] this change in the normal will produce the appearance of a wrinkled surface. This procedure is commonly referred to as bump mapping.

## 2.6 Bump Mapping

Traditionally when a surface is rendered the normal is recovered from the objects at the point required and used without any change applied to it before using it in rendering. However manipulation of the normal before using it in the lighting calculations can change the appearance of the object. In Simulation of Wrinkled Surfaces[3], James F. Blinn describes how the perturbation of normals can give a rendering extra detail not initially inherent in the object. Usually to add extra detail into the model extra polygons or more shapes would be used to describe detail, but in the case where the detail is that of small changes over the surface this would be impractical. But Blinn notes that the reason we notice these difference when rendered is the "effect on the direction of the surface normal". These permutations can be stored as an additional texture or can be generated at run time.

## 2.7 Turbulence

The turbulence function described by Perlin is a function for giving the "appearance of turbulent flow". The function is described as follows:

$$turbulance(p)=\sum_i \left|\frac{1}{2^i} noise(2^i p)\right|$$

Where $i$ is summed from 0 to when $\frac{1}{2^i}$ is smaller than the relative pixel size and $p$ is a point in 3-Space.

The notion of solid texture was extended by Perlin and Hoffert to the notion of "hypertexture"[4]. Hypertexture is a shape filling function like a solid texture but where the density is changed throughout the volume. With a solid texture the density is assumed to be a constant 1.0 but a hyper texture can vary in the range [0, 1] where at 0 the texture no longer exists. Hypertexture can be used to produce textures that have a shape of their own, thus creating a shape that exists within the bounding volume rather than contributing directly to the colour produced by the texture. Since this project is focused on the creation of solid textures that are used to texture given shapes the use of lower density volumes produced by hypertexture is unnecessary. However the paper provides other methods for manipulating the noise function and so the examples given are useful.

## 2.8 Reflections and Refractions

The reflection of an incident ray is such that the angle of incidence is the same as the angle of reflection, where both angles are measured away from the normal.

Reflections come paired with refractions by the way of the Fresnel reflection law which governs how the angle of incidence from the incoming ray will effect how strong the reflection is of the surface. The proportion of the light reflected and refracted is governed by the following formula formula.[5]

$$R=\frac{\sin^2(\theta_1-\theta_3)}{\sin^2(\theta_1+\theta_3)} \qquad T=\frac{\sin 2\theta_1 \sin 2\theta_3}{\sin^2(\theta_1+\theta_2)}$$

These are respectively the proportion of reflection and transition by a ray of light. Where $\theta_1$ is the angle between the incident ray and the normal; $\theta_2$ is the angle between the normal and the reflected ray and $\theta_3$ is the angle between the transmitted ray and the normal when the normal is facing downwards.

Due to the properties of reflected light $\theta_1=\theta_2$

By Snell's law $n_1\sin(\theta_1)=n_2\sin(\theta_3)$ where $n_1$ is the refractive index of the material before the refraction and $n_2$ the refractive index of the material after the refraction.

Schmidt and Budge[9] state that refraction within ray tracing can be complicated by having the boundaries of two refracting objects intersect with each other under a union combination. In this instance the ray must be correctly refracted into each different refractive index and out when necessary. This is as opposed to merely using the refractive index of the first intersection and assuming the value doesn't change until the object has been left.

# 3 Requirements Specification

Since the goal of this dissertation is to implement a working system for rendering solid texture with CSG ray tracing the requirements specification is an essential step. This will guide the overall development of the system in order to create a final product that correctly achieves the goals needed. In order that the following requirements are rigidly defined the following definitions and formats have been used to ensure that the system does not veer away from the goal.

There are three different categories of requirement that this document deals with they are Functional, Non-functional and User. The functional requirements are the intended behaviour of the system. The non-functional requirements are to do with the underlying system focusing on the performance of the system. The user requirements are the requirements that relate to the user interaction with the system.

Requirements come in three types. Rated by necessity they are the following.

- Absolute requirements are essential and must be met in the final system. The testing phase of the project will check that these requirements have been met.
- Strong requirements are those that while not essential to the system are highly recommended and should be strove to be met. They should be treated as extensions that improve the system.
- Optional requirements are non essential parts of the requirements specification and should be treated as extensions to the base system.

## 3.1 Definitions

With the aim of making the following requirements unambiguous the following definitions apply.

- System – The system is the final program produced and the dependencies that the program relies on.
- Fail – The instance when the system is forced to terminate by the Operating System. Such an instance may occur when there is an unhanded system error.

## 3.2 Requirement format

To clarify the requirements the key words "must","should", "may" are used to mean following

- must – Defined as an absolute requirement
- should – Defined as strong requirement.

- may – Defined as a purely optional requirement

## 3.3 Analysis

The requirements laid out present a system that is based on what is deemed required to  implement the features discovered over the course of the literature review.

## 3.4 Functional

### 3.4.1 Combined Solid Geometry (CSG)

CSG must be used as the modelling format so that it is easy to view the effect of the solid texture, since objects may be subtracted from each other. The system must implement union and subtraction to view the effects of these functions. The system may implement intersection, symmetric difference and merging but aren't necessarily required functions.

### 3.4.2 Objects

The system must have a sphere as an object to be manipulated in the scene as this is the base object. The system should have other objects that allow easier viewing of the different effects.

### 3.4.3 Lighting

The system must have direct and ambient lighting as these are the basic light types. The ambient light allows the colour of each material to be scene from all points on the object. The direct light shows the effect of lighting format based on the normal of the object.

The other lights to be implemented into the system should be spot lights and point lights as both these lights light the objects in a differing system

The system must be able to handle the diffuse and specular lighting from different types of light sources. The system should be capable of casting shadows from the light sources. These shadows may be coloured by a transparent material.

### 3.4.4 Output

The system must output using a 24bit colour palette since this will give full tonal range as can be outputted by most display devices.

The system must be capable of displaying the current progress of the image being rendered. The system must also output the rendered image to image files as specified by the user.

#### 3.4.5 Reflection and refraction

The system must be able to perform reflection on reflecting surfaces. The system must be able to perform refractions. The system should be able to handle refractions that happen involving objects that intersect each other. Managing to keep the refraction index correct though out.

#### 3.4.6 Materials

The system must be able to handle simple solid textures. The system must be able to handle tree based textures with differing numbers of branches. The system must be able to store the values of the colour, reflection value, refraction index, shininess and the normal perturbation. Each of these values must be used for the rendering purposes.

### 3.5 Non Functional

#### 3.5.1 Fast operating code

The code should be written with the aim of running at an optimum speed. Optimisations made must not be platform dependent.

#### 3.5.2 Multiple thread capable

Since ray tracing is a highly parallel operation, the implementation of tree based textures must not interfere with this. To this end the system must be written as multi threaded code and the system tested on a multi processor machine to make sure that the system is thread safe and operates correctly in this environment.

#### 3.5.3 System Errors

All errors encountered by must be handled internally without causing the system to fail.

The system must inform the user of the error encountered giving details of what occurred. The system must also shut down cleanly upon the error being encountered.

#### 3.5.4 Cross platform operation

The system must be able to be used on different operating systems and platforms. The code must only need to be recompiled to complete the porting process for a compiled language or work transparently on an interpreted language.

### 3.6 User

The user must be able to stop the system rendering without the system failing or not responding.

### 3.6.1 Input

The scene files used by the program must be in easy to read format. This is to enable the user to know what is going to be rendered before the rendering begins, since the rendering operation may take a long time to occur.

The user must be able to place the objects and lights in the scene and be able to construct the CSG tree and material trees required using only the scene file format and not requiring other means to enter scenes.

The user should be able to change the resolution of the render as well as the name outputted by the system.

The user should be able to change the location, orientation and nature of the camera used to view the objects.

The user must be able to view the progress of the render as the rendering is happening. This should be represented as a live render so the user can cancel the render mid progress upon seeing a mistake in the scene to be rendered.

When the user makes a non recoverable error in the language the file input system must tell the user the nature of the error while also giving the location of the error. Likewise when the user makes a recoverable error the input system must tell the user the nature and location of the mistake and what has been done to correct it.

# 4 Design

The system was designed as eight separate parts which could be seen as a collection of functions or as classes. The eight sections are the following: Operational flow, Renderer, Objects, Lights, Materials, Camera, File Input and Output. These eight were designed around several data types which were seen as fundamental to the implementation of the complete system. This design begins with describing the data types used followed by sections outlined above.

## 4.1 Data Types

The data types deemed to be essential to the implementation of the system was the following: Colour, Vector, Matrix, Ray, CSG tree and Ray hits.

As per the requirements Colour needs to be 24bit this means that it should be represented as its component parts. In order to provide accuracy for the final image at 24bit it is necessary to implement colour internally at a higher precision, using 24bit for the output formats.

Vectors are a fundamental part of 3d spacial calculations. The system will use homogeneous coordinates with the vectors taken as row vectors.

Rays are also key in ray tracing. Rays will represent the lines of the form $\vec{A}+\alpha\vec{B}$ where $\vec{A}$ is the starting point of the ray; $\vec{B}$ is the direction of the ray and α is distance down the line.

The matrix used in the system is the transformation matrix which is used for changing what space vectors are in. Since vectors are treated as row vectors the transformation matrix is viewed as the following:

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

where the elements are used as follows:

- $r_{11} r_{22} r_{33}$ are used for scaling in the x, y and z directions respectively
- $p_x p_y p_z$ are used for perspective transformations
- $t_x t_y t_z$ are used in translation in the direction (x,y,z)
- $r$ elements are used for rotation.

## 4.2 CSG Tree

The CSG tree is how the different objects are connected together. It is a binary tree where each point is either a boolean method for connecting the branches below or an object. The different boolean methods of connecting branches are as described in the literature review.

The objects defined are in object space with its centre at the origin, because of this, when the CSG tree is being traversed the ray must be transformed into the model space for each model created. This can be done by having matrices along each branch of the CSG tree, coupled with their inverses for when vectors have to be transformed from objects space to world space.

Since the matrices will not have to be changed at run time they can be accumulated to the leaves of the CSG tree by multiplying the matrices at the leaves by every one above them in the tree. Then only storing this matrix and its inverse at the leaf which will transform the ray from world space into object space with only one matrix multiplication.

The ray can be cast into the CSG tree with these matrices

## 4.3 Ray hits

When the ray is cast through the CSG tree it will hit multiple objects. This can be represented as an in order list of values of alpha associated with the object that was hit. In order to perform the different CSG operations every element of this list will have to store the two things. Firstly the alpha value at which the ray entered or exited the relevant object; and secondly whether there is solid or not along the list until the next element.

Using these values it is possible to calculate the functions of the CSG tree as follows.

- Union – Every element of the ray hits lists from the branches below are added but when one branch of the ray hits has no solid until the next in the list this is changed to indicate that the other list still has elements
- Intersection – Elements are only combined onto the intersected list when there is solid on both branches of the CSG tree
- Subtraction – Elements on the left branch are added if there is no solid on the right and elements on the right branch are added with the nature of their solid until next element reversed
- Symmetric Difference – Elements on both branches are added but if there is solid on one the other is added in reverse.

As mentioned above there are eight sections of the system that were designed to work together.

### 4.4 Objects

Objects all should provide the same function so that their implementation is interchangeable with each other. The objects each need to be able to tell if the ray manages to hit them and if so what the near and far hit points are on the object. This will allow the ray hits list to be able to tell between which points there is solid. The objects must also be able to give the normals to the point required in object space. All the objects must, in object space, be centred at the origin and axis aligned since this eases the calculation of both hit points and normals. The objects chosen to be included are sphere, cylinder, plane and cube. The first two both have spherical surfaces which means that the the normals change continually over the space, this means effects such as reflection and refraction can be tested for their use of the the normal. The sphere and cube are both bounded objects which means that their use within the CSG tree is easier to see.

### 4.5 Lights

There are four different types of light which are used in order to have opportunities to change the how the scene is lit so that effects such as bump mapping and shadowing can be correctly checked. The four different light sources are the four basic types of light namely point lights, spot lights, distant lights and ambient lights. They vary as described below:

- Distant light – A light that is considered infinitely far away and as such the light doesn't suffer from fall off and all the rays from the light are parallel
- Point light – A light that has a location which is within the scene, the light has a fall off of $\frac{1}{d^2}$ where d is the distance between the light and the item being lit
- Spot light – A light that has both a location and a direction. The light has a pair of angles associated with it. The first describes the maximum angle away from the point the spot light is directly pointing at for which the spot light has full brightness. The second describes the outer cone of the spot light where it doesn't produce any light. In between angles one and two the light falls off smoothly.

- Ambient light – A light that has no location or direction but hits every object within the scene with the same intensity.

## 4.6 Materials

The materials used within the system are the 3d solid textures as described in the sections before. The material trees are capable of returning several values for their properties, these are the following: colour, shininess, the normal perturbation, the reflection factor and the refractive index. The materials are set up into a tree structure which has to be traversed to get the details of the texture required. The materials used in the system each have between zero and three branches. The varying quantity of branches means that a variety of different effects can be achieved. The materials are as follows:

- Solid Colour – The base material, which has no children, all the values set by the material stay the same throughout the material.
- Sine Texture – The colour of the texture is interpolated between colours by the nature of the sine function, the normal perturbation varies as the normal of the sine function which is $\frac{-1}{\cos(x)}$
- Chequered Texture – The Chequered texture is a two branch tree texture that creates a Chequered pattern between the materials in such a way that the appearance is chequered across each axis.
- Rings – This material has a centre out from which two materials are switched between radially, producing rings in any orientated plane which parses through the centre
- Noise Texture – Using a procedural noise function the noise material branches between three materials. This is done such that two of the materials do not come into contact with each other.
- Turbulence – Turbulence adds procedural noise effected turbulent flow to the single material that it is connected to.

Since the materials are solid textures they exist within their own material space. Due to this each material in any tree needs to have a matrix to convert the point asked for from world space into the material space for the particular material. Unlike the CSG tree the matrices cannot be accumulated to the bottom of the material tree because the transformed point is required at each material that switches between multiple materials. If the matrices were accumulated to the end the switches performed would be incorrect. The perturbed normal returned from each material also has to be inversely transformed out of material space into world space.

## 4.7 Camera

The camera is made up of two parts the camera and the screen. The camera will be used as a point of reference to spawn the rays from performing the lens transformations implicitly rather than with a matrix. The screen is used to implicitly transform points from the screen space into a point on the screen in world space. Giving the ray a point to aim at.

## 4.8 Output

The output functions will provide what is needed to take the final rendered image and output it as an image file. Taking the 24bit colours from the rendered output and putting them into a file in the format required. The formats that are used are targa and BMP images. The output functions will name each file according to the name specified by the user adding the file suffix required.

Output will also handle the logging of the render giving exact details of the scene rendered.

## 4.9 File input and language

The file input functions read in the file requested by the user. The language used has been created for the system. The design of the language was governed by making it easy to manipulate the objects within and keeping the style of each element consistent.

The language was chosen to be a block basic language as used in POV-Ray[5] where the nested blocks would accurately represent the tree structure that is used in both the materials and the CSG tree. The language allows changing of every detail of the objects, lights, materials, output and the camera that the user requires.

The syntax for the language Blocks are denoted with curly brackets, values for properties are within round brackets,multiple values for properties are separated by commas and lines are ended with a semi colon.

The keywords used were chosen to be descriptive and to relate exactly to the items that they are related to.

To further aid the user the language has support for C style comments which can comment out a proportion of a line or multiple lines. The syntax for these is to have /* before the commented out block and */ afterwards. C++ style comments which can comment out the end of a line or a single line. The syntax for these is to have // before the comment, this lasts until the end of the line.

The ability to give a constant name to any material, tree, light or matrix block means that the user can reuse objects without having to reproduce the code. This eases the writing as complicated models containing many objects can be used multiple times without having to reproduce the same lengthy code multiple times. The syntax of naming a constant is to put the name before opening the block for the type of item the user would like to have a name attached. To use a previously defined constant the syntax is the constant name followed by a semi colon.

The ability to include other files allows the user to make use of constants defined in multiple files while also allowing the user to keep different models in different files, giving them the ability to keep the code clean. The syntax of file includes is include “file name”

Full details of the language are provided in Appendix B.

## 4.10 Renderer

The renderer section is where the elements described above are combined together to output the final image. The section is in two parts, the renderer and the progress display. The progress display shows either the portions of the image currently rendered on screen or a message to the console telling the user the current progress.

The render is a recursive ray tracer. For each pixel on the final output a ray is cast from the camera, through the screen into the world. For each object hit, the material at that point is taken. If the material is reflective the reflected ray is calculated then this ray fired into the scene using the same mechanism as used initially. The direction of the reflected ray is calculated by using the

To light the point a shadow ray is cast from each light, apart from the ambient light, to the object to discover if the first item hit along the ray after the light is the same as the ray that comes from the camera. By comparing whether the object hit is the same rather than checking if both rays hit the same point the potential rounding error is avoided.

If the light has hit the same object then the specular and diffuse contributions from the light source are combined.

The specular component of the light is calculated by taking the angle between the reflected direction of the light and the ray from the eye. This angle to the power of the shininess multiplied by the intensity completes the specular component.

The diffuse component is the angle between the light and the normal of the surface multiplied by the intensity, subtracting the specular component.

They are combined by multiplying the diffuse component by the colour of the material and then adding.

## 4.11 Operational Flow

The operational flow of the system is the main function that is first run that starts the other sections in the order that is required by the user. The steps that are run are the following steps:  Set up, file loading, rendering and output.  The steps are run in the order above.

The function of the different elements of the operational flow is described below

- Set-up - Sets options that the user asks for in running the system. Including the scene file to be loaded. The number of threads to use and the nature of the file outputs.
- File loading – Loads the scene from file and sets up the data structures needed by the system.
- Rendering – Renders the scene by casting rays into the scene to determine the object hit then calculating the lighting determined by the material set to the object.
- Output – Performs all the file output functions that the options dictate. Including output the rendered image to an image format and outputting a log of the render.

# 5 Implementation

This section will discuss the implementation of the system developed over the course of this project. Key features and some of the low level functions and structures will be discussed.

## 5.1 Language

The code was written in C++. The language was chosen for familiarity and for the speed. C++ allows the use of classes which means that items in the design that had multiple objects performing the same task in a different way could be implemented using polymorphism. This meant that abstract base classes could be used for many of the system elements that had multiple different types that had to work within the same system equally. C++ also allows code that is not in a class when it is not required. This meant that functions that performed together in a procedural manner were not required to be in the same class function as seen in languages such as JAVA.

C++ also allows operator overloading which was used to make the manipulation of the data types easier and meant that the code read correctly in a mathematical manner. Easing the work required to implement functions based on mathematical algorithms.

By using a graphical IDE to develop the system it was possible to group the functions and files by type while working on them meaning it was easier to implement the sections described within the design.

## 5.2 Resources

In order to implement cross platform multi threading SDL[7] library was used. This provided a quick and easy way to get multi threading code working on all systems without having to use platform specific code. SDL was also used to create the window and draw the pixels to the screen to show the progress of the render in a graphical manner.

In order to provide effective and complete implementations of fundamental data structures the standard template library(STL)[10] was used rather than using my own implementations which would take time and potentially be inferior.

## 5.3 Data Types

There are several data types which are used to perform the mathematics behind the different functions. These are vectors, colours, matrices and rays. Each was created as a class with overloaded operators for the different mathematical functions that is implemented. Below is a description of each of these data types and how how they are used:

- Vector – Comprised of three double precision floating point values. The vector allows addition, subtraction, unary minus, dot product and scalar multiplication. It also has a function to return a vector of unit length in the direction of the vector.
- Colour – Comprised of three double precision floating point values. The colour class allows addition, subtraction and multiplication by another colour and multiplication and division by a scalar. It also has a function to clip the values of the colour to between 0 and 1.
- Ray – Comprised of two vectors and a scale factor. The ray is made of the form $a+s\alpha b$ where a is the ray origin, b is a unit vector in the direction of the ray, s is the scale factor of the ray and α is a distance along the ray. The class has a function for returning the vector of the point a given α along the ray
- Matrix – Comprised of sixteen double precision floating point values. The matrix class allows multiplication by vectors, rays, other matrices and normal vectors. It also has a function to return the inverse of the matrix.

## 5.4 Collection classes

There are two classes in the system which are used as data transport devices, combining the data which is associated with each other. These are AlphaInfo and MatData. AlphaInfo combines the information from an entry or exit point from an object along the ray. MatData combines the information returned by a material.

## 5.5 Data Structures

The CSG tree is made up of CSGNodes which can either be of type CSGTree or CSGLeaf. The node type defines the basic functions that both the tree and leaf types have to implement. As described in the design and the requirements the functions that are general to each node is to accumulate matrices, march the ray through and to print itself out.

## 5.6 Ray hits

For each object in the CSG tree the entrance and exit distances from the origin of the ray are calculated. This is coupled to several other pieces of data in the AlphaInfo class. These are:

- bool rearFace - This stores whether this is the front or rear face of the given object. This is stored so that the normal of the rear surface can be inverted from the one that would otherwise be returned.

- bool solidTillNext – This stores whether there is solid existing between this and the next element of the vector. This is used both in the CSG boolean methods and finding the correct side of a particular object.
- const void *CSGnode – This a void pointer to the CSG leaf that the object concerned is connected to. It is a void pointer because the class is defined as part of the CSGNode which in turn uses it. Since this is a cyclical reference a void pointer is used.
- Bool ignore – This is used for objects within the scene that are not visible.

Rayhits uses the STL vector class to store the hits along the ray and to store the alphaInfo elements in a container class that is in order and the elements of which can be easily iterated through.

## 5.7 Multi threading

Multiple threading is achieved by using threads and mutexes which are supplied by SDL. Initially the number of thread the user specifies are spawned. The threads work on a value line which holds the number of lines called to be rendered by threads. Connected to the line value is a mutex which is what SDL uses to stop more than one thread accessing a value at the same time. The thread handler attempts to lock the mutex before starting, if the mutex is already locked by another thread the handler waits until the mutex is available to lock. Once the mutex is unlocked the handler reads the value, compares it to the final height of the render and then adds ten before unlocking the mutex.

By doing things in this way the line value stores the last rendered line asked for and so can be used as a progress meter.

Another mutex is used for the final rendered output. This output is a 2D array made of pointers to pointers. It was done this way as opposed to the flat array that works as a 2D array because it means that less time is spent in the mutex as all that needs updating is a pointer reference, compared to having to update every element of the array.

## 5.8 Rendering

The rendering process follows a set algorithm. Which is as described in pseudo code.

```
Create ray with camera and screen
Get first hit from ray
if (no hit)  return background
if (refractive) cast refraction ray through objects
if (reflective) cast reflective ray through objects
for each light in the list {
    if (light has a direction) {
        Get first hit from shadow ray
        if (the object hit by shadow ray same as ray hit from camera) {
            light object
        }
        if (no shadow hit) {
            light object
        }
    }  else {
        add ambient light
    }
}
return the combination of each component
```

By following this algorithm every light in the scene is used and the reflection and refractions are handled by the recursive nature of the raytracer.

### 5.9 Refractions

Refractions are a special case because they travel into the solid textures. Thus the material must be marched through to discover if the material changes along the path of the ray. Unfortunately this portion of the implementation could not be completed due to time constraints. But the framework for the work is present in the code. Standard refraction works as expected.

### 5.10 Display Window

The display window uses the final rendered image to display the progress. By iterating through the 2D array it is easy to discover which lines are yet to be rendered as the pointers are null. This speeds up the drawing process as the system doesn't have to check the entire line.

In the event that the window fails to open, due to SDL error or due to user not wishing it to be open the same function handles the console output.

# 6 System Testing

The system testing is split into two sections the first is for testing how well the system conforms to the requirements set out in the requirements specification. The second section tests the system using black box testing to discover if the expected out come is achieved.

## 6.1 System conformity

Since the requirements specification was split into three sections this conformity check will be split into the same three sections. It is recommended to use the requirements specification as a guide to this section.

### 6.1.1 Functional

#### Combined Solid Geometry (CSG)

The final system implements combined solid geometry as the modelling format. Implementing all five of the different CSG operations.

#### Objects

The final system implements spheres as well as having an abstract data type for objects so that different object types may be created. Additional objects enhanced the modelling ability of the system.

#### Lighting

The final system implements all four different lights through an abstract light data type.  The system handles the diffuse and specular components from each light. The system also casts shadows from objects where the lights would cast shadows. The system doesn't handle coloured shadows.

#### Output

The system uses outputs to two different 24Bit image formats, targa files and Bit Map Pictures. The system also used a 24Bit pixel format to draw the progress window to the screen. The system is capable of turning off the progress window and still displaying the rendered progress.

#### Reflection and refraction

The system is capable of performing reflections and simple refractions. Unfortunately the system is unable to handle refractions that involve the union between two objects. Though it can handle the other combination types. The refraction index is maintained through out.

#### Materials

The system is capable of handling solid coloured materials. The system has within it materials that have between one and three different branches. The materials hold the colour, reflection value, refraction index, shininess and normal perturbation as required. Though the latter isn't made full use of and directional noise was not used.

### 6.1.2 Non Functional

#### Fast operating code

There are no platform dependant code optimisations. The system is designed to run quickly but no explicit rendering optimisations were made. However the implementation of intersection means that users can put a bounding volume on the left branch that will speed up rendering.

#### Multiple thread capable

By using SDL threads the system is capable of running multiple threads to render the image. The threading implementation means that very little time is spent in mutex locks and so rendering time is decreased by using multiple threads.

#### System Errors

The system has been designed with the intention that no system errors occur due to illegally run code segments. This was done to negate the need for error catching which can slow the system down. For functions where errors may occur the system closes down with quick exits.

#### Cross platform operation

By using C++ the code is portable, using the STL and SDL libraries does not decrease the portability of the code as both are available on vast ranges of systems. Though within windows the SDL link library must be included with the executable to operate.

### 6.1.3 User

The system allows the user to quit the process at any time by quitting the application. Also while the window is active it is possible to stop the system by pressing escape. Though both quitting functions must wait for the delay to finish before being acted upon

#### Input

The scene files are designed with ease of use in mind. Using the file format the user is capable of placing the objects and lights within the scene and create the trees as required. The scene file language is capable of handling all the inputs possible. The resolution can be set in the scene file. The name used to render the object can be set in the scene file and on the command line.

While the scene file language allows the camera and screen to move neither works correctly in the final system.

The system displays the render live if the user wishes and if the window isn't shown a progress meter is still displayed.

When the user makes a mistake the system tells the user the nature of the problem, the file the problem occurred in and what line the problem occurred. The system also explains where a potential problem such as unclosed comments or strings started.

## 6.2 Software Testing

The system was tested with many different scene files. But as there are so many different combinations of objects and lights all the possible options could not be tested.

Each of the command line arguments were tested. Each performed exactly as expected for each of the files loaded. The window appeared when required and didn't when the user requested it didn't.

The system was tested on two machines:

- An Athlon 64 4000+ with 1Gb of ram running Windows XP.
- A 4 way Xeon 1.8 Ghz with 2Gb of ram running WindowsServer 2003

  The above machine was run in one and four threads mode.

The scenes rendered were

- matDemo.scene – This scene is a simple one with four objects
- Dice.scene – This scene has the highest object count at with 52 objects. It has a reflecting floor that uses the sine material
- Trees.scene – This scene contains few objects, but contains more lights than the above.
- extremeMaterialTest.scene. - This scene contains a reflection and refraction

Each run was done with the following options. -w which means that the window display isn't used.
The results were as follows

## 6 System Testing

| ***Scene*** | ***Athlon 64*** | ***Xeon 1 thread*** | ***Xeon 4 threads*** |
|---|---|---|---|
| MatDemo | 6237ms | 15061ms | 12594ms |
| Dice | 21097 ms | 58077 ms | 43477ms |
| Trees | 28491ms | 79059ms | 59720ms |
| EMT | 19610ms | 53051ms | 40756ms |

Unfortunately when rendering matDemo.scene the output would crash the system upon attempting to write the output file. The reason behind the error happening with this file and not others is unknown. The error also occurred in all but the dice render for the xeon with 1 thread. I am unaware of the reason behind this bug as it did not happen during development.

# 7 Conclusion

The goal of the project was to create a ray tracer that conformed to the specification as outlined in section three. The final system meets all the essential requirements except is more error prone than initially thought during development.

The successful parts of the project are due to making good use of the features of C++ to implement the system, which meant development could occur at a swift pace. Using SDL and the STL made implementation very simple.

## 7.1 Future Work

Future work on the system can be done to complete the objective of a completely working refraction index. This would mean that the materials would work as fully as intended

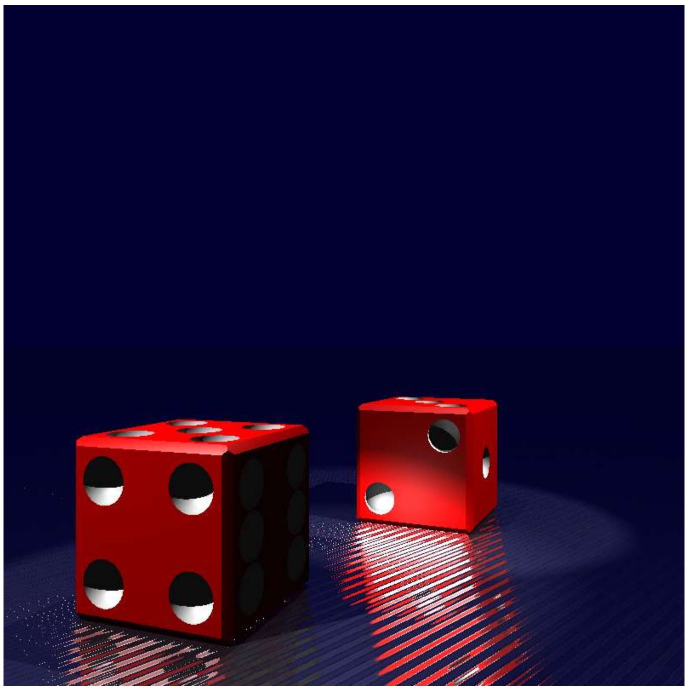
# 8 Bibliography

1. PERLIN, K. 1985. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques,* p287-296. 1985
2. PERLIN, K, 2002. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques, July 23-26, 2002, San Antonio, Texas*
3. BLINN, J. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques,* p.286-292, August 23-25, 1978
4. PERLIN, K. AND HOFFERT, E. 1989. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques,* p253-262. 1989
5. Reflection and refraction of light (Fresnel Formulas) http://physics.nad.ru/Physics/English/rays_txt.htm (last accessed: may 15th 2005)
6. POV-Ray – The persistence of vision ray tracer 2005 http://www.povray.org/ (may 15th 2005)
7. SDL – Simple Direct media Layer. 2005. http://libsdl.org/ (last accessed: may 15th 2005)
8. siggraph – ACM siggraph's website. 2005 http://www.siggraph.org/ (last accessed: may 15th 2005)
9. SCHMIDT, C AND BUDGE, B. Simple nested dielectrics in ray traced images. *In Journal of Graphics Tools, volume7 issue 2* p1-8, 2002
10. SGI Standard Template Library Programmers Guide. 1994. http://www.sgi.com/tech/stl/ (last accessed: may 15th 2005)

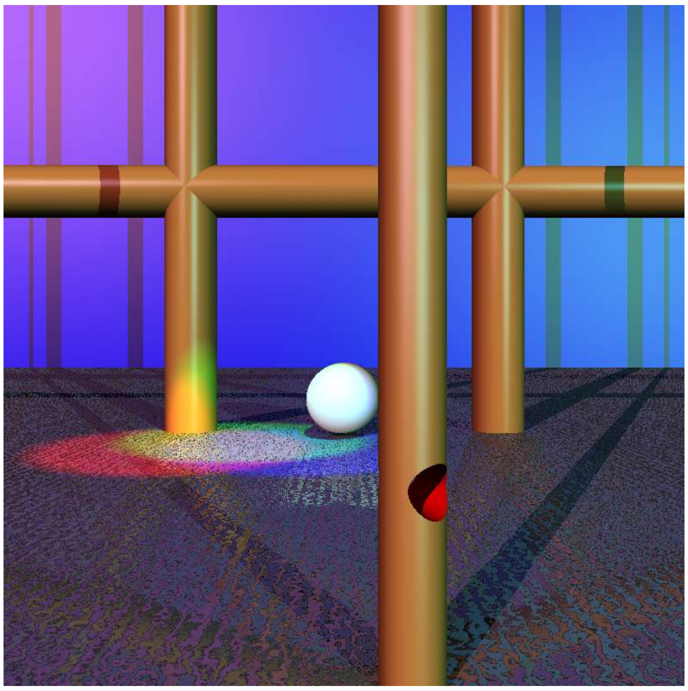
# 9 Appendices

## A Sample Renders

Output from Dice.scene

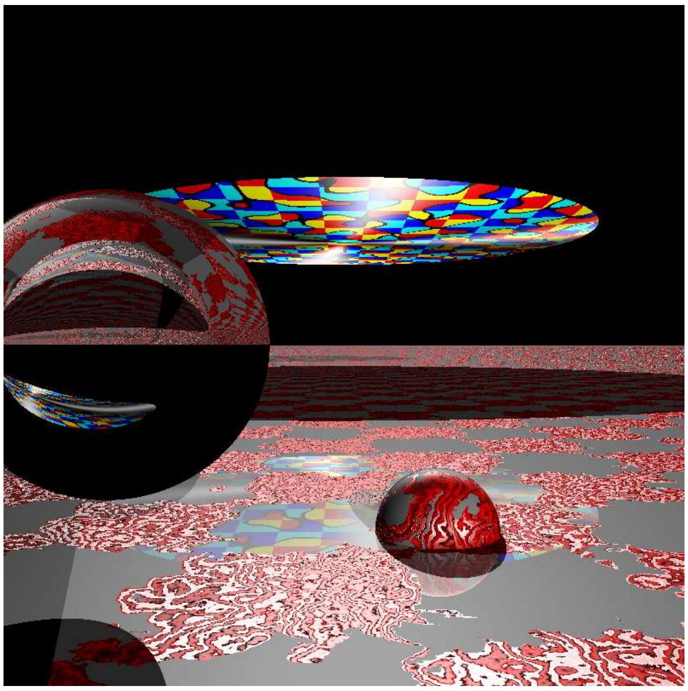


Output from trees.scene



## A Sample Renders

output from ExtremeMaterialTest.scene



# B Scene File Format

The scene file format is a block structured format. The language is case sensitive with all the words in the language being in lower case. The locations all are based on the origin and every angle is expressed in radians. The blocks are denoted by curly brackets. Blocks look like the following

```
objects {
        The block's contents resides here.
}
```

Values for properties are written by putting the property to be changed followed by round brackets containing the value. The property is finished by having a semi colon on the end of the line. For example

```
shininess(5);
```

For properties that have multiple values these values are separated by commas. For example

```
direction(1,0,0);
```

There are two methods for inputting comments into the files these are the following:

// comment

where the comment last for the entire line and

/* comment */

where the comment exists within the two symbols.

The main scene file is made up two main sections and two smaller optional sections. The two main ones are objects and lights, the two optional are camera and output.

### CSG tree: objects

The CSG tree created between the last objects block in the file will be the one used to render the output. The CSG tree is constructed using one of the CSG boolean operations. The CSG operations work with two blocks a left and a right block. The format for a CSG operation looks like the following

```
union {
        The left branch is described here
} with {
        the right branch is described here
}
```

The available CSG operations are union, merge, subtraction, intersection and symdiff. The last of which is short for symmetric difference.

#### Transformations

Along the branches of the CSG tree transformations can be placed which effect every element down the branch. Transformations are created with a transform block. Transformations have several functions to manipulate them they are the following:

- translate(x,y,z); - Translate in the direction of the vector (x,y,z)
- scale(x,y,z); - Scale with x,y and z scale factors.
- rotx(θ) – Rotate about the x axis by θ degrees.
- roty(θ) – Rotate about the y axis by θ degrees.
- rotz(θ) – Rotate about the z axis by θ degrees.]

The transform block can also have an inverse block which inverts the functions of the functions within the block before applying them to the current transformation.

#### The objects

The roots of any CSG tree are the objects that make up the scene. The objects all are defined in object space with set default values. The objects in the system along with their defaults are presented below.

- sphere – The sphere by default has its centre at the origin and has a radius of 1. The radius can be changed with radius(r);
- cube – The cube by default has its centre at the origin, is axis aligned and

each face is one unit away from the centre.

- plane – The plane by default has its centre at the origin, has its normal along the y axis and has a thickness of 0. The thickness can be changed with thickness(x);
- cylinder – The cylinder is an infinitely long cylinder with a radius of 1 and by default stretches along the y axis. The radius can be changed as the sphere with radius(r);

### The materials

By default every object has a default solid colour texture, of which the colour and shininess can be altered with colour(r,g,b); and shininess(s); To add a material to an object the material block must be defined within the object. A material block can be made up of the following material blocks. They come in two types tree blocks and leaf blocks. Tree blocks are made up of multiple blocks, depending on the material. Where as leaf blocks are not made of other blocks but have the following common properties.

- reflection(x); – Where x is how much is reflected compared to lit as a fraction. Default is 0
- refraction(x); - Where x is what the refraction index of the material is. This is in relation to space which has an index of 1. Default is no refraction.
- shininess(x); - Where x is how shiny and so how dense the specular highlight higher numbers indicate shinier objects. Default is 10.

The two leaf blocks in the language are

- solid - A pure colour material which can have the colour changed with colour(r,g,b);
- sine – A material made of two colours. The material interpolates between the colours across the x axis by the nature of the sine wave, which also defines the normal of the material.

The tree blocks are the following

- turbulence – This material takes one other, it adds turbulent flow to the values of the materials below it in the material tree.
- checkers – This material switches between two materials in checkers along each of the major axes. The material has a changing period of 1.
- rings – This material switches between two materials radially with a centre at the origin with a period of 1.

- noise – This material switches between three materials. This uses noise functions to generate a number between [-1,1]. For the period [-1,-0.05) the first material is returned. For the period (0.05,1] the second material is returned. For the period [-0.05,0.05] the third material is returned.

To apply a transformation to a material a transform block must be in the block which is being transformed.

## The Lights : lights

The lights in the system all have the following two properties colour and intensity. The colour can be changed with colour(r,g,b); and the intensity by intensity(x);. The four lights in the language are the following. The colour defaults to white for every light, while the intensity is default at 1 for all lights except for ambient which it is 0.05.

- direct – A directional light which has a direction. The direction can be changed with direction(x,y,z); which describes the direction in world space the light aims. The default direction is
- point – A point light which has a location which can be changed with location(x,y,z);. The intensity of the light at a point is defined as $\frac{1}{d^2}$ where d is the distance of the point from the light.
- spot – A spot light which has a location, a direction and two angles. The location and direction can be changed in the same way as the point and direct lights respectively. The angles can be changed with angles(a,b); Where a is the inner angle where the light has full intensity and b is the angle from which the light has no intensity. Between a and b the light falls off smoothly.
- ambient – The ambient light only has the default values. But the intensity by default is 0.05 as it doesn't cast shadows.

## Constants

Each of the block types described above can be assigned to a constant name. To name a block the format is demonstrated by the following:

```
sphere ball {
        // the sphere's properties
}
```

This will assign the name ball to the sphere. This attaches all the properties within the block to the name sphere. The name ball cannot be attached to a different object after this. To use a saved constant the constant name must be placed within any block where a constant of that type is suitable. Constants can be used in the following locations.

- Transformation – can be used transformations and trees.
- Objects or trees – can be used in trees
- lights – can be used in lights
- materials – can be used in materials and objects.

The syntax to use a constant is demonstrated by the following:

```
union {
   ball;
} with {
   // other branch
}
```

Which uses the ball in object space and unions it with the elements of the right branch of this union.

## The Camera : camera

The camera block describes the location and details of the camera. The camera has the following properties:

- location(x,y,z); - Sets the location of the camera, defaults to (0,0,0). Unfortunately currently the camera doesn't work properly moved from this location.
- focus(x); - Sets the focal distance of the camera defaults to 10.
- aperture(x); - Sets the aperture of the camera's lens, defaults to 0.2.
- points(x); - Sets the number of points to be pre calculated on the lens defaults to 1. The first of any calculated points will always be the very centre point. If set to 0 the camera will generate a different point on the lens for every ray.
- rays(x); - Sets the number of rays to be cast by the camera. Casting more rays than points will cause the camera to reuse older points.

The camera block also has the screen block. The screen block has location(x,y,z); and size(x,y); The location of the screen defaults to (0,0,1) and in the current version cannot be moved. The size of the screen in world space is described by size(x,y).

### The Output : output

The output block has two properties, the resolution of the final rendered image and the name of the outputted file. The resolution is changed with resolution(x,y); where the x and y are on screen sizes. The name of the output file is changed with name(file name); Where file name is a  string and must be contained within a pair of double quote as the example shows:

name(“file”);

### Background

Finally in the main body of the file the background colour, which is rendered when no object is hit. The format of this is background(r,g,b);

# C Command line arguments

The format of the command line for the program is the following.

```
RayTracer.exe [-bdflstvw? -j threads -o output] scene
        -b  output BMP files
        -d  tells not to render just parse scene. Sets verbose
        -f  displays a full screen window progress
        -l  saves a log of the render as 'output'.log
        -s  stops displaying warnings
        -t  output TGA files
        -v  verbose output
        -w  stops the display window appearing
        -?  display this help
        -j  chose number of threads
        -o change the output from the one used in the file load
```

## D Source Code

Due to the length of the source code it will not be reproduced here. A copy of the source can be found at
http://students.bath.ac.uk/ma1pm/raytracer/raytracer.tar.bz2 until June 2005