CS488 Project - Extended Ray Tracer - Manual

1 Program Information

1.1 Running the program:

`./rt <lua-file.lua>` will render the scene described in lua-file.lua and output an image as indicated in the lua file.

Optional parameters to `rt` are as follows:

- `-a <[0-3]>`
  Anti-alias level; defaults to 0 (no anti-aliasing); higher numbers produce nicer images but increase render time.
- `-d <[0-10]>`
  Maximum recursion depth for secondary rays cast for reflection and refraction; defaults to 0.
- `-f`
  Disable incremental file writing. Defaults to enabled, partial image is written to disk after every five percent of completion.
- `-n`
  Disable hierarchial bounding volumes – for testing only, slows ray-tracer significantly.
- `-s <[0-10]>`
  Grid dimension for shadow rays cast at non-point light sources; defaults to 1 (very poor quality soft shadows).
- `-p`
  Use vertex normal interpolation to produce smoothly rendered meshes.

1.2 Output

The program generates a progress indicator, printing the current percent complete at every interval of five percent. If incremental file writing is enabled then the program also writes the current partial image to disk at these points.

Upon completion, the program prints the following statistics:

- Render time in seconds
- Number of primary rays generated
- Number of shadow rays generated
- Number of reflected rays generated
- Number of refracted rays generated

- Total number of rays generated
- Number of ray-primitive intersection tests performed
- Average number of primitive intersection tests per ray
- Number of ray-bounding box intersection tests performed
- Average number of bounding box intersection tests per ray

1.3 New Lua Commands

The following new global (gr) Lua commands are supported by the program:

- `cylinder(<name>, <base_radius>, <top_radius>, <height>)`
- `cone(<name>)`
- `torus(<name>, <tube_radius>, <major_radius>)`
- `plane(<name>)`
- `csg(<name>, <operation - [uid]>, <node1>, <node2>)`
- `rr_material(<diffuse>, <specular>, <shininess>, <reflective>, <refractive>, <index>)`
- `bitmap_texture(<texture_file>)`
- `checkerboard_texture(<colour1>, <colour2>, <coord_system - [lw]>, <scale>)`
- `simplenoise_texture(<colour1>, <colour2>, <coord_system - [lw]>, <scale>)`
- `woodgrain_texture(<colour1>, <colour2>, <coord_system - [lw]>, <scale>)`
- `marble_texture(<colour1>, <colour2>, <colour3>, <coord_system - [lw]>, <scale>, <stripes_per_unit>, <octaves>)`
- `planelight(<position1>, <position2>, <position3>, <colour>, <falloff>)`
- `fullmesh(<verts>, <textures>, <normals>, <vert_faces>, <text_faces>, <normal_faces>`

Also, the member functions `set_texture(<texture>)` and `set_bump(<texture>)` were added to the material type in Lua.

1.4 Organisation

All source files are located in A5/src, the executable is located in A5/src, A5/data and in A5 with the README file. The data files located in A5/data include 21 test scripts to test the various features of my raytracer along with resource files and the files for my final scene. Also in A5/data is the "movie" directory containing an animation and the scripts to generate it. Finally there is a "website" directory in A5/data containing the presentation website as well as many rendered images.

2 Code Map

algebra.hpp, cpp
Contains code for use and manipulation of vectors, points, colours, matrices as per assignment 4, also contains code for rays, added in for the project.

hit.hpp, cpp
Definition files and helper functions for tracking intersection information.

image.hpp, cpp
Interface for working with the PNG image format, as provided in assignment 4.

light.hpp, cpp
Definition and helper functions for point and planar light sources.

lua488.hpp
Lua libraries includes, as in assignment 4.

main.cpp
Obtains command line arguments and kicks off parsing of scene file.

material.hpp, cpp
Contains code for determining the colour/shading of a point on an object given diffuse and specular properties as well as texture and bump maps.

mesh.hpp, cpp
Mesh intersection code.

perlin_noise.hpp, cpp
Implementation of standard 3D Perlin noise, based on Ken Perlin's reference implementation [Perlin, 2002].

polyroots.hpp, cpp
Quadric, cubic and quartic root solvers, as in assignment 4.

primitive.hpp, cpp
Intersection code for all non-mesh primitive types, as well as the intersect code for bounding boxes.

rt.hpp, cpp
Main raytracing algorithm, casts primary rays, checks for occlusion (shadows), and casts secondary rays recursively. Also performs anti-aliasing.

scene.hpp, cpp
Main data structure for storing nodes in DAG/tree. Does hierarchial bounding volumes, transformations, and initial preprocessing.

scene_lua.hpp, cpp
Interface between lua and C++, new Lua functions defined here.

texture.hpp, cpp
Computes colours for intersection points using either bitmaps or procedural solid textures.

3 Implementation

3.1 Additional Primitives

Intersection with new primitives is performed by substituting the ray equation ($p = p_0 + t * \overrightarrow{d}$) into the implicit equation for the primitive and solving for t. [Glassner]

Relevant code for new primitives is located in `primitive.cpp` and `primitive.hpp`.

Some equations were obtained from
`http://www.cl.cam.ac.uk/Teaching/2000/AGraphHCI/SMEG/node2.html`.

Cylinder

The following Lua command was added to support cylinders:

```
cylinder(<name>, <base_radius>, <top_radius>, <height>)
```

where `<name>` is an identifier for the primitive, `<base_radius>` is the radius at the end of the cylinder located on the x-y plane, `<top_radius>` is the radius of the cylinder at $z = height$ and `<height>` is the height of the cylinder.

The implicit equation for the canonical infinite cylinder oriented along the z-axis is $x^2 + y^2 = 1$. This is modified to be $x^2 + y^2 = m^2(z - c)^2$ where m is the change in radius from top to bottom, and c is a displacement along the z-axis used to obtain the correct section of the infinite volume. The gradient vector of this function is used to compute normals around the surface of the cylinder.

The ends of the cylinder are capped with discs defined by $x^2 + y^2 <= r^2$ where r is the radius at the end in question.

Intersection proceeds by first testing the discs and then testing the surface if less than two hits were obtained when testing the discs.

Cone

The following Lua command was added to support cones:

```
cone(<name>)
```

where `<name>` is an identifier for the primitive.

The implicit equation for the canonical double cone oriented along the z-axis is $x^2 + y^2 = z^2$. The gradient of this function is used to calculate normals on the surface. The cone is bounded at $z = 1$ and $z = 0$ to produce a single finite cone, and the end is capped with the disc defined by $x^2 + y^2 <= 1$ at $z = 1$.

Intersection proceeds by first testing the disc, then the surface.

Torus

The following Lua command was added to support tori:

```
torus(<name>, <tube_radius>, <radius>)
```

where `<name>` is an identifier for the primitive, `<tube_radius>` is the radius of the tube, and `<radius>` is the radius from the origin of the torus to the centre of the tube.

The implicit equation for a torus located on the x-y plane is $(\sqrt{x^2+y^2}-R)^2+z^2=r^2$ where R is the radius of the entire torus and r is the radius of the tube. Substitution of the ray equation into this equation produces a large quartic function which is solved with the quartic root solver supplied in assignment 4. See `http://www.cl.cam.ac.uk/Teaching/2000/AGraphHCI/SMEG/footnode.html#foot252` for the full equation. The gradient of the standard torus equation is used to compute normals.

Plane

The following Lua command was added to support planes:

```
plane(<name>)
```

where `<name>` is an identifier for the primitive.

This constructs the plane $y = 0$ bounded by -1 and 1 on both the x and z axes. This is a useful primitive for creating walls, ground, ceiling, etc. Intersection is simply testing for intersection with the plane $y = 0$ and then testing the intersection point against the bounds of the plane.

3.2 CSGs

The following Lua command was added to support CSGs:

```
csg(<name>, <operation - [uid]>, <node1>, <node2>)
```

where `<name>` is an identifier for the CSG; `<operation>` is a single character indicating the operation to perform: union, intersection, or difference; and `<node1>` and `<node2>` are the left and right operands of the CSG operation respectively. `<node1>` and `<node2>` can either be primitives or CSG nodes. This implies that trees of CSG operations can be constructed.

Code relevant to CSGs can be found in `scene.cpp`, `scene.hpp primitive.cpp`, `hit.hpp`, and `hit.cpp`.

A new node type was added to the DAG which has an operation, as well as exactly two children - the operands of the operation. This node extends the basic scene node from assignment 4, and the GeometryNode class of assignment 4 was altered to extend the CSGNode

class. This creates an easy way to limit the children of a CSGNode to be only CSGNodes or GeometryNodes.

Intersection with CSGs required the primitive intersection routines to be modified to return all hits. Two structs were created to store this information:

```
enum HitType { ENTER, EXIT }; // is the hit entering or leaving the object
typedef struct hit_struct {
    HitType type; // type of hit (entering or leaving the object - CSG)
    double sval; // distance along vector
    Vector3D normal; // normal at hit point
    Point2D uv; // uv coord at hit point
    Vector3D u; // u vector
    Vector3D v; // v vector
} Hit;

typedef struct hr_struct {
    Ray3D ray; // ray to intersect with stuff
    GeometryNode* object; // the hit object
    std::list<Hit> hits; // all hits in object, sorted from nearest to furthest
} HitRecord;
```

The HitRecord struct is passed up and down the tree and values are filled in where appropriate. At the CSG node, the operation is performed by first obtaining intersection information from each child and then by comparing the lists of hits.

For CSG union, the hit lists are merged into one list sorted by sval, and then the list is processed to save all outer <ENTER,EXIT> pairs. Entrance to the union code requires that at least one child returns hits. For CSG intersection a state machine is used to save all pairs <ENTER,EXIT> that occur within an outer pair. Entrance to intersection requires that both children return hits.

Difference is performed by noting that diff(a,b) is just intersect(a,complement(b)). Since we are working in just one dimension (the distance along the ray), the complement of b is computed by first finding all hits in b, then by flipping the types and normals of those hits. Finally, an extra hit is placed at the beginning and end of this list: one before all of the hits (sval = -1) and another that comes after the further of the last hits of both a and b. Then the intersection code is used to process these new lists.

Hit lists are carefully modified and constructed to avoid degenerate states. CSG nodes require that hit lists have balanced ENTER and EXIT hits. If this is not the case, the CSG intersection code gives up and decides that there is no hit. A common cause of this is when a ray starts inside of a primitive.

The HitRecord struct contains a GeometryNode representing the object that was hit by the ray. In CSGs multiple objects may be hit. The intersection code chooses the best match of the node's two children for which node to pass up the tree. In difference, the left child al-

ways goes up; in union, the closest hit goes up; and in intersection, the further object goes up.

All primitives except meshes support CSG operations. Meshes have partial, best-effort support, but their functionality is not guaranteed.

3.3 Hierarchical Bounding Volumes

Relevant code is located in `scene.cpp`, `scene.hpp`, `primitive.cpp`, and `primitive.hpp`.

Hierarchical bounding volumes are created during the preprocessing of the DAG. Preprocessing involves creating multiple copies of various DAG nodes to produce a strict tree allowing instancing. Also, the final transformation matrix is stored in each GeometryNode to reduce the number of matrix multiplications during the ray tracing stage.

When a primitive is created, an axis-aligned bounding box is defined for it in the primitive's local coordinate space. During preprocessing, a GeometryNode obtains the local coordinates of the box of its primitive and then transforms the box to world coordinates. Internal nodes retrieve the world coordinates of the bounding boxes of all of their children and then compute the world coordinates for a bounding box which contains all of the child boxes. By storing bounding boxes in world coordinates, ray-bounding box intersections become quicker because the ray does not need to be transformed before intersection.

Bounding box intersection is performed at each internal node in the strict tree, and if the ray does not hit the box, the children of the node are not processed.

Fast bounding box intersection is implemented using the method described in Williams et. al. This involves precomputation of some properties of the ray (1.0/direction and the sign of the components of the direction). The precomputation helps improve performance when a single ray is intersected with many bounding boxes.

A bounding box is stored as two points, one with minimum values in all components and one with maximum values. When transforming to a new coordinate system, all 8 corners must be translated and then new minimum and maximum coordinates selected from the set.

The flag `-n`, if given to the program, disables bounding box intersection tests.

3.4 Soft Shadows

The following Lua command was added to create a planar light source:

```
planelight(<position1>, <position2>, <position3>, <colour>, <falloff>)
```

where `<position1>` is the base position, `<position2>` is a different corner of the quadrilateral, `<position3>` is the opposite corner and `<colour>` and `<falloff>` are the same as in the standard light command.

A planar light source is defined by 3 points, and stored as a point and two vectors. I chose the three point representation because 3 points are always coplanar in 3-space. This guarantees that the light is a plane. The first point is used as a base point, and then two vectors are computed, $v_1 = p_2 - p_1$ and $v_2 = p_3 - p_1$. Any position, p, in the light source can be computed as $p = p_1 + (s * v_1 + t * v_2)$ for $s, t \epsilon [0, 1]$. This way of defining the light allows for any parallelogram to be defined as a light source. It also avoids accidentally defining a non-coplanar set of points.

To achieve the soft shadow effect, the shadow ray code was altered to return a real number between 0 and 1 indicating how much of the light source is visible. Multiple rays are cast towards various points on the light and the percent that hit the light indicates the percent of the light that is visible.

To determine the points to cast the rays towards, the shadow ray code queries the light for a list of points. The light computes a list of evenly spaced points across the surface, and jitters each one by a small amount in both directions (v_1 and v_2). This jittering produces noise instead of banding, and the noise can be reduced by increasing the number of shadow rays or by anti-aliasing the image.

3.5 Reflection and Refraction

The following Lua command was added to support reflective and refractive surfaces:

`rr_material(<diffuse>, <specular>, <shininess>, <reflective>, <refractive>, <index>)`

where `<diffuse>`, `<specular>`, and `<shininess>` are the same as in the standard material command and `<reflective>` is the percent, [0,1], of light reflected by the surface, `<refractive>` is the percent, [0,1], of light refracted by the surface, and `<index>` is the index of refraction.

Relevant code is located in `rt.cpp` and `material.hpp`.

At each intersection point, if the surface is reflective, a new ray is cast in the direction of the incoming vector reflected about the normal at the intersection point. This is computed as $\overrightarrow{r} = -\overrightarrow{v} + 2(\overrightarrow{v} \cdot \overrightarrow{n})\overrightarrow{n}$ as described in the CS488 course notes.

Similarly if the surface is refractive, a new ray is recursively cast in the direction dictated by Snell's law: $\overrightarrow{r} = \frac{\eta_i}{\eta_r}\overrightarrow{v} - (\cos\theta_r - \frac{\eta_i}{\eta_r}\cos\theta_i)\overrightarrow{n}$ where $\cos\theta_r = \sqrt{1 - (\frac{\eta_i}{\eta_r})^2(1 - \cos^2\theta_i)}$ [Hearn and Baker, 600].

Once a colour is determined for reflected and refracted light, the colour is multiplied by the specular colour of the surface, since reflection and refraction are specular effects.

The contributions of the reflected and refracted light are then scaled by the amount of light the material reflects and refracts. Then the scaled reflected contribution is added to the direct contribution. This total is scaled by $1.0 - refract\_amount$. Finally, the scaled refracted contribution is added to the the scaled total. Ambient light is added to the direct light

before combining with reflected and refracted light. I have no reference for this combination because I could not find any. This was achieved through trial and error.

3.6 Texture Mapping

Texture maps are stored in materials. The lua command `set_texture(<texture>)` was added to the material type in order to set the texture of a material. When colour is being computed at an intersection point, the material checks if it has a texture, and if so it retrieves it's main colour from the texture. Two types of textures were created, bitmap (2D) and solid (3D).

Relevant code is located in `material.cpp`, `texture.hpp`, `texture.cpp`, `perlin_noise.hpp` and `perlin_noise.cpp`.

Bitmap Textures

The following Lua command was added to create bitmap textures:

`bitmap_texture(<texture_file>)`

where `<texture_file>` is the name of a raster image file located in the current directory.

On intersection a primitive also calculates the 2-dimensional, UV coordinates of the intersection point. This point is passed to the material when calculating colours, and the material passes it to the texture if it has one. The Bitmap texture uses the UV coordinates (which are in the range [0,1]) to index into the specified image file. All primitives except meshes can have a bitmap texture applied to them. Bitmap texturing on CSGs is undefined.

References for the following include [Shirley and Morley, pp. 95-97] and
`http://www.cl.cam.ac.uk/Teaching/2000/AGraphHCI/SMEG/node2.html`

UV coordinates for a sphere are calculated using the parametric equation of a sphere with $origin = (x_c, y_c, z_c)$ and radius R:

$x = x_c + R\cos\phi\sin\theta$
$y = y_c + R\sin\phi\cos\theta$
$z = z_c + R\cos\theta$

which gives:

$\theta = \arccos(\frac{z-z_c}{R})$
$\phi = \arctan(\frac{y-y_c}{x-x_c})$

and can be converted to UV coordinates with:

$u = \frac{\phi}{2\pi}$
$v = \frac{\pi-\theta}{\pi}$

since $(\theta, \phi)\epsilon[0, \pi] \times [-\pi, \pi]$, assuming that one adds 2π to ϕ if it is negative.

UV coordinates for a plane are simply defined as the distance along the plane in each of the two defining directions (generally two non-parallel edges). The UV coordinates for a cube are the same, just repeated for each face.

UV coordinates for a cylinder are broken into two parts. First, the discs (located on plane parallel to x-y plane, with radius R) can be defined by distance from the centre (x_c, y_c) and rotation angle from the starting point:

$r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$

$\theta = \arctan(\frac{y-y_c}{x-x_c})$

the the UV coordinates are:

$u = \frac{r}{R}$

$v = \frac{\theta}{2\pi}$

assuming that the disc is parallel to the x-y plane and that if θ is less than 0, 2π is added to it first. Second, the surface of the cylinder can be defined by distance along the surface (of length height) and the angle rotated around the surface:

$h = \frac{z}{height}$

$\theta = \arctan(\frac{y}{x})$

assuming that the cylinder is located along the positive z-axis, then the UVs are:

$u = \frac{\theta}{2\pi}$

$v = h$

as long as 2π is added to θ first if it is negative.

UV coordinates for cones and cylinders with different top and bottom radii are equivalent to that of the standard cylinder, as long as they are located along the positive z-axis.

UV coordinates for a torus are based on the parametric equation for a torus with radius R and tube radius r located in the x-y plane:

$x = (R + r\cos\phi)\cos\theta$

$y = (R + r\cos\phi)\sin\theta$

$z = r\sin\theta$

where θ is the angle around the major circle and then ϕ is the angle around the tube. So:

$\phi = \arcsin(\frac{z}{r})$

$\theta = \arctan(\frac{y}{x})$

then the UVs are:

$u = \frac{\theta}{2\pi}$

$v = \frac{\phi}{2\pi}$

as long as both θ and ϕ are first adjusted by 2π if they are negative.

Solid Textures

Solid textures are implemented as functions defined over space. Four new Lua commands were created defining different solid textures:

`checkerboard_texture(<colour1>, <colour2>, <coord_system - [lw]>, <scale>)`

`simplenoise_texture(<colour1>, <colour2>, <coord_system - [lw]>, <scale>)`

`woodgrain_texture(<colour1>, <colour2>, <coord_system - [lw]>, <scale>)`

`marble_texture(<colour1>, <colour2>, <colour3>, <coord_system - [lw]>, <scale>, <stripes_per_unit>, <octaves>)`

where the `<colourX>` parameters are the colours involved in the texture, `<coord_system>` is a single char, w or l specifying whether the texture should be computed in local or world coordinates for any particular object, `scale` is a scaling factor and the marble texture has two other defining parameters described below.

Solid textures can be applied to all primitives, as well as to CSG objects, due to the fact that they are independent of the object being textured.

The checkerboard texture simply defines alternating rectangular volumes of colour oriented along the z-axis.

The other three textures make use of Perlin noise. The noise is implemented as described in [Perlin 2002] and using the reference implementation located at `http://mrl.nyu.edu/~perlin/noise`. The noise works by computing the function at any point (x,y,z):

$$n(x,y,z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor+1} \sum_{i=\lfloor y \rfloor}^{\lfloor y \rfloor+1} \sum_{i=\lfloor z \rfloor}^{\lfloor z \rfloor+1} \Omega_{i,j,k}(x-i, y-j, z-k)$$

where:

$\Omega_{i,j,k}(u,v,w) = \omega(u)\omega(v)\omega(w)(g_{i,j,k} \cdot (u,v,w))$

and:

$\omega(t) = 6t^5 - 15t^4 - 10t^3$ as long as $t \epsilon [0,1]$.

$g_{i,j,k} = G(\phi(i + \phi(j + \phi(k))))$

for G, a precomputed array of N vectors and $\phi(i)$ gives a number in the range [0,N] from a precomputed permutation (I used the one given at `http://mrl.nyu.edu/~perlin/noise`). See also [Shirley and Morley, pp. 81-82].

The effect of this function is as follows: for any point, do seven linear interpolations of the corners of a lattice containing the point, combining the vectors of G randomly as you do it and using the function $6t^5 - 15t^4 - 10t^3$ which has 0 at 0 and 1 of its first and second derivatives. This produces smooth transitions in the randomness with very low memory and computational cost. Once the noise function is defined, it can be perturbed in various ways to get interesting textures. The noise function returns a value in the range [-1,1].

The simple noise texture defined above is basically just a scaling of the noise, and adjusting it to the range [0,1] and then doing linear interpolation between the two

defined colours.

The woodgrain texture is the function:

$t(p) = noise(p) * 20.0$

$t = t - floor(t)$

and then linear interpolation of the two colours using t. This function was found at `http://freespace.virgin.net/ hugo.elias/models/m_perlin.htm`.

The marble texture uses the idea of a turbulence function to get different sizes in the features of the noise, the turbulence function used for the marble is:

$$n_t(p) = \sum_{i}^{M} \frac{n(2^i p)}{2^i}$$

The marble takes parameters for stripes per area and octaves, in addition to a scaling factor. The point is scaled base on the stripes per area, then it is passed to the turbulence function and octaves is used for the value M above. The resulting value is scaled again by the scaling factor. Finally, t is computed as:

$t = 2|\sin(stripes\_per\_area\pi x + result)|$

where x is the x component of the point, and result is the value computed through the turbulence function above. This gives a t value between 0 and 2, if it is between 0 and 1 then the colour is the result of linear interpolation from colour2 to colour3, otherwise it is the result of linear interpolation from colour1 to colour2. By varying the scale, stripes per area, and octaves one can achieve different looks in the marble texture. This function was defined in [Shirley and Morley, pp. 91-93].

3.7 Bump Mapping

Bump maps are stored in materials and are just textures which are used to perturb normals instead of colours. In theory, any texture can be used, however the program currently only supports bitmap textures as bump maps. The lua command `set_bump(<texture>)` was added to the material type in order to set the bump map of a material. When the colour of a hit is being computed, the material checks if it has a bump map, and if so then it uses it to perturb the normal before continuing.

Normal perturbation is computed as described in [Blinn 1978] and also using material covered in class. Relevant code is located in `material.cpp`, `hit.hpp` and `primitive.cpp`.

To support Blinn's method for bump mapping, surface tangent vectors must be computed at the each surface intersection point to be used as a basis for perturbation of the normal. These vectors are computed at the same time as the intersection with the primitive, and are stored in the Hit struct described in the CSG section. The vectors are computed using the partial derivatives of the parametric equations of the surface. That is if the surface is defined by:

$x = a(u, v)$

$y = b(u, v)$

$z = c(u, v)$

for parametric functions a, b, and c, then given an intersection point (u_0, v_0), the tangent

vectors in the u and v directions are the partial derivatives:

$\overrightarrow{P_u} = (\frac{\delta a}{\delta u}(u_0, v_0), \frac{\delta b}{\delta u}(u_0, v_0), \frac{\delta c}{\delta u}(u_0, v_0))$

$\overrightarrow{P_v} = (\frac{\delta a}{\delta v}(u_0, v_0), \frac{\delta b}{\delta v}(u_0, v_0), \frac{\delta c}{\delta v}(u_0, v_0))$

Once these vectors are computed in the local space of the primitive, they are transformed into world coordinates by the transpose of the inverse of the node's transform matrix, similar to how the normals are transformed. Next, we need to find the partial derivatives of the bump map at the uv coordinate. This can be approximated as:

$F_u = \frac{B(u_0+\epsilon,v_0)-B(u_0-\epsilon,v_0)}{2\epsilon}$

$F_v = \frac{B(u_0,v_0+\epsilon)-B(u_0,v_0-\epsilon)}{2\epsilon}$

where $B(u, v)$ retrieves the average colour of the bump map at (u, v) and ϵ is some reasonable sampling value such as $\frac{1}{map\_width}$. Finally, the perturbed normal can be calculated at the point as:

$\overrightarrow{N'} = \overrightarrow{N} + \overrightarrow{D}$

where:

$\overrightarrow{D} = (F_u(\overrightarrow{N} \times \overrightarrow{P_v}) - F_v(\overrightarrow{N} \times \overrightarrow{P_u}))$

and $\overrightarrow{D}$ is normalised before adding it to $\overrightarrow{N}$. For more details, see [Blinn 1978].

Since I use average colour as the domain of the bump mapping function, it is best to create bump maps as grey scale images.

3.8 Phong Shading for Meshs

Vertex normals can be specified for triangular meshs, and if they are present, they are interpolated across the surface of the face to produce smooth shading of the mesh (as opposed to faceted). The lua command `fullmesh(<name>, <generate_vertex_normals>, <verts>, <textures>, <normals>, <vert_faces>, <text_faces>, <normal_faces>)` was added to allow the specification of vertex normals and texture coordinates (UVs) for meshs. Additionally, the mesh command can be instructed to attempt to generate vertex normals for the mesh. Note that this will only approximate the proper vertex normals, since a mesh is normally the result of a tessellation of some curved surface. Normals generated on tessellation will almost always be better since they would be based on the curve and not the tesselated data.

Normal interpolation is accomplished by the standard Gouraud method of interpolation. The area of the full face is precomputed and stored with the vertex data. On intersection, the areas of the three contained triangles are computed and the appropriate proportion of each normal is summed to produce the final normal vector. This vector is then used for illumination calculations.

The area of a triangle in three-space, specified as three vertices p_1, p_2, p_3, can be computed as:

$A(\Delta) = \frac{1}{2}|(p_2 - p_1) \times (p_3 - p_1)|$

since, for vectors $\overrightarrow{u} = p_2 - p_1$, $\overrightarrow{v} = p_3 - p_1$, and θ, the angle between them:

$|\overrightarrow{u} \times \overrightarrow{v}| = |\overrightarrow{u}||\overrightarrow{v}||\sin\theta|$

Vertex normal computation is done by accumulating a sum of face normals for all faces adjacent to a vertex and then normalising the result.

4 Possible Improvements/Extensions

There were several features I wanted to do, but did not get time to implement. These include:

- Solid bump mapping – need to determine a way to compute partial derivatives of 3D texture, not just 2D.
- Diffuse reflection/refraction – similar to soft shadows, need to cast multiple rays at reflection and refraction points jittered slightly, but all in the general direction of reflection/refraction.
- Adaptive anti-aliasing – speed up anti-aliasing by examining variance of colour in subpixels of a current pixel and only subdivide further if the variance is too high. Can also do jittering to help produce better results.
- Interpolation of bitmap textures – currently, the image index is computed as $(\lfloor u * width \rfloor, \lfloor v * height \rfloor)$, and depending on the image and primitive this can produce unpleasant results. A better solution would be to use bilinear or higher order interpolation of the colour at the UV coordinate based on nearby intergral values.

5 Acknowledgements

I acknowledge that the fast box intersection code located in primitive.cpp, in the function BoundingBox::fast_intersect is based on the code given in [Williams, 2005].

I acknowledge that the code contained in perlin_noise.cpp and perlin_noise.hpp is based on the reference implementation of Perlin noise given by Ken Perlin (for the purpose of creating a standard in noise functions) at `http://mrl.nyu.edu/~perlin/noise`

I acknowledge that I did not create the mathematical functions used to manipulate the noise function to create the effects seen in my marble and wood grain solid textures. The creators are referenced in the appropriate sections of this manual.

I acknowledge that I did not write the majority of the code contained in the files polyroots.cpp, polyroots.hpp, algebra.hpp, algebra.cpp, image.hpp and image.cpp.

I acknowledge that the number of spelling and grammar mistakes in this document is probably rather large. Apologies.

6 Bibliography:

Blinn, J.F., "Simulation of Wrinkled Surfaces." In Computer Graphics, Vol. 12, no. 3, 1978. pp 286-292.

Department of Computer Graphics, "CS488/688 Course Notes", Spring 2005.
Glassner, Andrew S. [ed], "Introduction to Ray Tracing", Academic Press Limited, 1989.

Hearn, Donald and M. Pauline Baker, "Computer Graphics with OpenGL, Third Edition", Prentice Hall, 2003.

Hill, Francis, "The Pleasures of 'Perp Dot' Products." In: Graphics Gems IV, IBM, 1994.

Perlin, Ken and E. Hoffert, "Hypertexture." In Computer Graphics (Proc. of ACM SIGGRAPH '89), 1989.

Perlin, Ken, "Improving Noise." In Transactions on Computer Graphics (Proc. of ACM SIGGRAPH '02), 2002.

Perlin, Ken, "Improved Noise Reference Implementation", 2002. URL:
`http://mrl.nyu.edu/~perlin/noise`

Shirley, Peter and R. Keith Morley, "Realistic Ray Tracing, 2nd Edition", AK Peters Limited, 2003.

Sunday, Dan, "Intersections of Rays, Segments, Planes and Triangles in 3D"
URL: `http://www.softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm`

Wiliams, Amy, Steve Barrus, R. Keith Morley, and Peter Shirley, "An Efficient and Robust Ray-Box Intersection Algorithm." In Journal of Graphics Tools, Vol. 10, No. 1:49-55, 2005.