Manual

Contents

1 Running the Program **1**

2 Using the Features **1**
2.1 Anti-Aliasing 1
2.2 Multithreading 1
2.3 Cone and Cylinder Primitives 1
2.4 Texture Mapping 2
2.5 Bump Mapping 2
2.6 UV Parametrizations 2
2.7 Remapping 3
2.8 Constructive Solid Geometry 3
2.9 Reflection, Refraction and Transparency 4
2.10 CSG Union and Medium Interfaces 4
2.11 Soft Shadows 5
2.12 Caustics 5

3 LUA Commands **6**

4 Program Options **7**

1 Running the Program

The ray tracer is driven by the same style of LUA scene scripts as in assignment 4. It now accepts several additional commands, which are laid out in section 3. The program is run on the command line, and it accepts as arguments an optional series of command-line flags and name-value pairs (see section 4) followed by a series of LUA scene scripts to run. If no scripts are specified, the program attempts to run `scene.lua`. As in assignment 4, images can be produced from scene files using the `gr.render` function.

While the program is rendering a scene, it outputs information about elapsed time and progress.

2 Using the Features

This section describes how to use the features of the ray tracer. Features which were mandatory components of assignment 4 will not be discussed unless they have seen changes since assignment 4.

2.1 Anti-Aliasing

Adaptive anti-aliasing is used in this ray tracer. It reduces 'jaggies' by performing rendering calculations multiple times for pixels containing colour discontinuities. There are two parameters to adjust: the sensitivity to discontinuities (ie, the number of pixels which are selected for anti-aliasing) and the number of samples used for anti-aliasing a selected pixel. In practise, anti-aliasing is inexpensive given a reasonable threshold. See also the `--aa-grid` and `--aa-threshold` program options.

2.2 Multithreading

This ray tracer takes advantage of thread-level parallelism to achieve significant performance boosts. To achieve optimal performance, use the `--threads` program option to have the program use a number of threads which is approximately equal to the number of logical cores on the machine.

2.3 Cone and Cylinder Primitives

It is possible to create cone and cylinder primitives via the `gr.cone` and `gr.cylinder` commands. These primitives are hierarchal and both have heights 2 and radii 1. The material to be attached to the primitive must be passed in when it is created.

2.4 Texture Mapping

Texture mapping is used in this ray tracer so that users may 'map' pictures onto primitives. In particular, materials with associated textures have their diffuse colour coefficients determined by the texture.

A user may create a texture using the `gr.texture` function. The function accepts a single string argument which is either the name of a procedural texture or the path of a PNG image. The only procedural textures that are supported are 'redcheck', 'greencheck', 'bluecheck', and 'whitecheck', which are checker patterns of varying colours. After a texture is created, it may be attached to a material using the `material:set_texture` method.

Each primitive has a parametrization by which PNG images are mapped to them; see section 2.6.

2.5 Bump Mapping

Bump mapping is used to perturb the surface normals of primitives according to a user-specified map. Doing so can produce impressive images by having the direction of light bounces give the impression of detailed shapes; but bump mapping is usually much less difficult than modeling the detailed shapes themselves, and rendering simple shapes with bump maps is certainly less expensive than rendering detailed ones.

Users may create bump maps using the `gr.bumpmap` function. The function accepts a single string argument which is either the name of a procedural bump map or the path of a PNG image. The only procedural bump map which is supported is the 'ripple' bump map, which simulates the ripple effect of water which has been disturbed. After a bumpmap is created, it may be attached to a material using the `material:set_bumpmap` method.

Image bump maps must be grayscale PNG images. Normal perturbations are drawn from bump map images via the image gradients; specifically, normals are perturbed in the direction of positive gradients. Thus: on a rendered primitive, areas corresponding to darker areas of the bump map will appear more elevated.

Each primitive has a parametrization by which PNG images are mapped to them; see section 2.6.

2.6 UV Parametrizations

Texture and bump mapping require parametrizations of each primitive so that two-dimensional images can be mapped to the three-dimensional primitives. For this purpose, the ray tracer parametrizes the primitives with respect to rectangular images in the following ways.

For cube primitives, the image is divided into twelve squares in three rows of four. Each face on the cube is linearly mapped to one of these squares. If we number the squares from 1 to 12 – top to bottom, left to right – then square 6 is mapped to the top of the cube, and squares 2, 5, 7, 8, and 10 are mapped to the other faces of the cube as if they had been wrapped around it. The parametrizations of individual faces are consistent with this wrapping metaphor.

For sphere primitives, an image texture is mapped to the primitive as if it had been curled into a top- and bottom-less cylinder around the sphere and then projected onto the sphere from that position.

For cylinder primitives, the lower half of the image is wrapped around the non-planar face of the primitive. The upper half of the image is divided equally into two squares on the left and right side. The maximal circles inside these squares are mapped to the top and bottom of the cylinder, respectively.

For cone primitives, the lower half of the image is wrapped around the non-planar face of the primitive. The maximal circle inside the upper half of the image is linearly mapped to the planar face of the cone.

2.7 Remapping

It often happens that one has an image that they wish to map to only part of a primitive as a texture or bump map. There is limited support for doing just that through the `texture:remapt` and `bumpmap:remap` methods, which accept single strings and reparametrize the images based on the type of remapping that the string refers to. Examples of supported remap types are:

- `cubetop` – The entire image is mapped to the top of the cube.

- `cubefront` – The entire image is mapped to the front of the cube.
- `cubebot` – The entire image is mapped to the back of the cube.
- `cyltop` – The entire image is mapped to the top of the cylinder.
- `conetop` – The entire image is mapped to the cone's planar surface.

2.8 Constructive Solid Geometry

The idea behind constructive solid geometry is to provide only simple primitives, like the sphere, cube, cone, and cylinder primitives supported by this ray tracer, and allow users to combine them in order to create more complex shapes. To this end, constructuve solid geometry provides three basic operations on hierarchal nodes: union, difference, and intersection. The ray tracer exposes these operations in the form of the `gr.union`, `gr.difference`, and `gr.intersection` functions, which all require exactly two node operands and return CSG primitives.

The **union** function combines two hierarchal nodes into a single CSG object. This had no visible effect except insofar as the resulting object can be used in other CSG functions.

The **difference** function takes two hierarchal nodes and returns a CSG node which contains the non-empty volume of the first node which does not intersect with the second node. In other words, the difference function *subtracts* its second operand from its first. In the resulting node, there may be newly exposed surfaces of the first operand which were formerly adjacent to volume which intersected with the second subtracted operand. These surfaces take on the properties (ie, the materials and normals) of the intersecting volume of the subtracted shape.

The **intersection** function returns a CSG node which contains only the volumes within which its children intersect. Every surface in the resulting node will correspond with a surface in one of the children volumes; and it is from these corresponding surfaces that the properties of the resulting CSG node are derived (ie, the material and normals).

In this ray tracer, CSG union nodes have the special property that they yield information about interfaces between component volumes. See section 2.10.

2.9 Reflection, Refraction and Transparency

The reflectivity, refraction and transparency of materials are determined only by their specular colour coefficients and refractive indices.

- Any material with null specular coefficients is opaque and non-reflective.
- Any material with nonzero specular coefficients and a refractive index which has not yet been set is opaque and reflective. The intensity and colour of reflected light depends only on the incoming light and the specular coefficients of the material.

- Any material with nonzero specular coefficients and a refractive index which has been set is transparent (and necessarily reflective as well). The intensity and colour of transmitted and reflected light is determined by the incoming light, the specular coefficients, and the Fresnel equations.

2.10 CSG Union and Medium Interfaces

A problem with modeling refraction in ray tracers is that there is no obvious way to support refraction between two non-air mediums. For example: to model liquid in a glass, you might think to simply place a cylinder of liquid directly adjacent to the sides of the glass object. This will not work, though, because ray tracers typically ignore collisions that occur within a certain distance of the source, and because there is no way for the program to know whether there is meant to be air between the two objects.

The problem is solved in this ray tracer via the CSG union function. Namely, CSG union nodes yield information about interfaces between their component volumes. If you wish to create an object (such as a glass of liquid) which contains interfaces between non-air mediums, you must do so using the union function.

The exact semantics of this aspect of the union function are a bit complex, but an intuitive understanding can be achieved via the following principles:

- Whenever the volume of the left union operand ends within ϵ units of where the right operand's volume begins (where ϵ is some hard-coded constant), that is taken to be an interface between the volumes.
- Whenever the two operands intersect, the second operand takes priority – ie, the second operand determines the volume's material.
- Whenever the volume of the second operand begins inside the volume of the first operand, that is (obviously) an interface between the two volumes.

2.11 Soft Shadows

The naive approach to shadows is to use point lights and to colour pixels one of two colours based on whether they are occluded. This is often undesirable, as in the real world shadows give way gradually to light; the edges of shadows are blurry. This ray tracer supports these "soft shadows".

To enable soft shadows, you must simply make your light into a spherical area light using the `light:set_radius` method.

Soft shadows are achieved via repeated sampling. Sample density, and thus shadow quality, can be managed through the `--shadow-grid` program option.

2.12 Caustics

The backward ray tracing employed by typical distributed ray tracers cannot easily simulate a plethora of caustic effects that light often produces in the real world. In order to achieve these effects, this ray tracer uses a method

called 'photon mapping'. Photon mapping employs a preprocessing phase which involves repeatedly tracing randomly directed photons from each light through the scene.

Photon mapping is disabled by default. If you wish to produce caustic effects, you must provide a nonzero integer argument to the `--caustic-num-photons` program option, which determines how many photons are shot during the preprocessing step (a reasonable value for typical scenes is 1000000). Also important is the `--caustic-num-neighbours` program option, which determines how many photons are used in each colour estimate (default: 100).

In general, the quality of caustic effects will increase as both parameters increase (though the `--caustic-num-neighbours` parameter should always be a small fraction of the other). Small values will result in undesirable noise.

You should also be aware that photon mapping is extremely sensitive to lighting parameters. The most relevant ones are the third falloff parameter for lights, which must not be zero, and the unit distance (see the `--unit-distance` program option). If photon mapping fails to produce reasonable results, check these parameters.

3 LUA Commands

The following is a listing of the LUA commands and methods supported by the ray tracer. For the sake of conciseness and clarity, commands which were mandatory components of assignment 4 will be ommitted (though they are also available).

- `gr.cone(name, material)` – Return an hierarchal cone with the specified name and material.
- `gr.cylinder(name, material)` – Return an hierarchal cylinder with the specified name and material.
- `gr.texture(str)` – Return the texture object described by `str`. `str` may be either the relative path of an RBG PNG image file or one of the strings `redcheck`, `greencheck`, `whitecheck`, `bluecheck`. See section 2.5.
- `gr.bumpmap(str)` – Return the bumpmap object described by `str`. `str` may be either the relative path of a grayscale PNG image file or the string `sinewaves`. See section 2.4.
- `texture:remapt(str)` – Remap the specified texture as described by `str`, which must be one of the strings `cubetop`, `cubefront`, `cubebot`, `cyltop`, `conetop`. See section 2.4.
- `bumpmap:remap(str)` – Remap the specified bumpmap as described by `str`, which must be one of the strings `cubetop`, `cubefront`, `cubebot`, `cyltop`, `conetop`. See section 2.5.

- `material:set_texture(texture)` – Have the material use the given texture object for its diffuse coefficients.
- `material:set_bumpmap(bumpmap)` – Have the material use the given bump map to perturb normal vectors.
- `material:set_ri(ri)` – Sets the refractive index of the object to `ri`. See section 2.9.
- `material:set_reflective()` – Makes a material reflective.
- `gr.union(c1, c2)` – Creates a constructive solid geometry node representing the union of `c1` and `c2`. See section 2.8.
- `gr.intersection(c1, c2)` – Creates a constructuve solid geometry node representing the intersection of `c1` and `c2`. See section 2.8.
- `gr.difference(c1, c2)` – Creates a constructive solid geometry node representing the difference between `c1` and `c2` (in other words, the resulting node is $c1 \setminus c2$). See section 2.8.
- `light:set_radius(rad)` – Sets the radius of the light to `rad` (see section 2.11.
- `gr.option(str)` – Pass an option string to the program. Option strings take the same form as command-line arguments. See section 4.
- `gr.set_miss_colour(colour)` – Sets the colour to be used when a ray fails to hit any objects in a scene.

4 Program Options

Program options are global values which affect the appearance of the image produced as well as the runtime of the program.

There are two types of options: flag (boolean) options, which are by default disabled and can only be enabled, and single-value options, which must always be accompanied by either integer or real values.

There are two ways to toggle flags or specify single-value options: via the command line and via the `gr.option` LUA command. On the command line, any number of strings (for flag options) or whitespace-separated string-value pairs (for single-value options) can be specified *before* the first scene file. The `gr.option` command accepts a single string which has the same form as a sequence of command-line arguments (note in particular that you can modify many program options through a single call to `gr.option`).

The following is a comprehensive listing of valid program options, in approximately descending order of usefulness.

- `--threads N` – Have the program use up to N threads. N should be the number of logical cores on the machine for maximum performance. Default: 6.
- `--aa-grid N` – Specify that the program should use an NxN grid when it performs anti-aliasing for a single pixel. Higher values may improve image quality but will reduce performance. Default: 3.
- `--aa-threshold R` – Specify the discontinuity threshold above which anti-aliasing should be performed on a pixel (see section 2.1). Higher values may reduce image quality while improving performance. Default: 0.5.
- `--caustic-num-photons N` – Specify the number of caustic photons to be shot during the caustic photon mapping preprocessing stage (see section 2.12). Higher values will reduce performance while improving caustic quality. See also `--caustic-num-neighbours`. Default: 0.
- `--caustic-num-neighbours N` – Specify the number of neighbours to be used in caustic radiance estimations (see section 2.12). Higher values will reduce performance while improving caustic quality (though this is dependent on the value of –caustic-num-photnos). Default: 100.
- `--shadow-grid N` – Specify that the program sould use an NxN grid when it computes soft shadows. Higher values may improve shadow quality but may reduce performance. Default: 4.
- `--energy-fudge D` – Have the photon map adjust the amount of energy used in photon shooting for every light proportionally to D. This is useful for situations when you want to abandon a physically realistic model for artistic reasons. In particular, you may wish to have objects produce caustic effects despite being far away from the relevant light. Default: 1.
- `--unit-distance D` – Specify the unit distance. Higher unit distances will increase the strength of lights with non-zero second and third falloff coefficients and will increase the brightness of caustics. This option is useful when dealing with scripts with very large or very small scenes (ie, scripts which scale their objects by very large or very small amounts). Default: 1.
- `--gi` – Enable global illumination via photon mapping.
- `--gi-num-photons` – Set the number of photons to shoot for the global photon map.
- `--gi-num-neighbours` – Set the number of neighbours to use in global illumination radiance estimates.
- `-t, --timing` – Print the runtimes of various phases of the program after it's done rendering a scene.

- `-s, --silent` – Don't print normal output. Will still print errors.
- `-S, --stats` – Print various statistics after the program is done rendering a scene.
- `-v, --verbose` – Print debug output.
- `-d, --debug` – Print debug output and suppress normal output.
- `--disable-aa` – Disables anti-aliasing. Equivalent to '–aa-grid 1'.
- `--disable-bv` – Disables mesh bounding volumes. Will likely reduce performance.
- `--draw-aa` – Colours pixels selected for anti-aliasing red.
- `--draw-caustic-map` – Draws caustics photons on the image rather than performing normal lighting calculations.
- `--draw-caustic-prm` – Draws red pixels on diffuse surfaces inside the range of the caustic projection map for just one light.
- `--minres` – Override scene scripts and always render images with resolution 100 by 100.
- `--lores` – Override scene scripts and always render images with resolution 256 by 256.
- `--midres` – Override scene scripts and always render images with resolution 512 by 512.
- `--hires` – Override scene scripts and always render images with resolution 1024 by 1024.
- `--rays` – Draw a sunny background where rays fail to intersect with objects in the scene.