CompSci715 Assignment 2, 2002

Ray Tracing & Curves and Surfaces

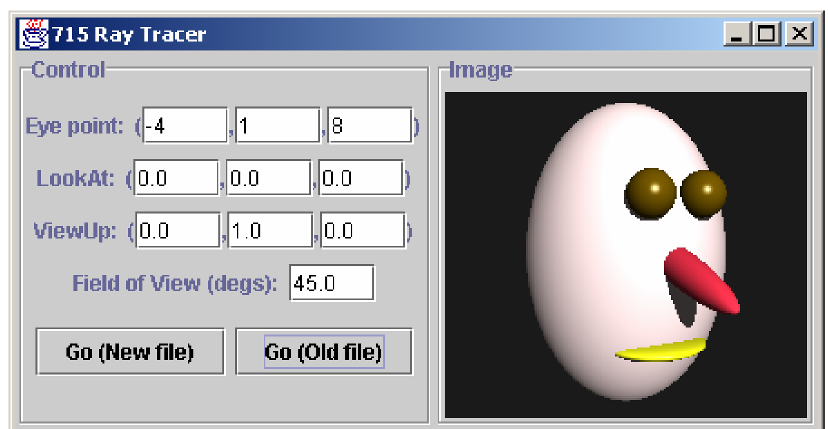
Due: 11:00am Thursday 22 August.

Getting Started

Download and unzip *Ass2Source.zip* from the web server. The contents of the folder *RayTracer* are for part A and the contents of the folder *TeapotLid* are for part B of this assignment. The *geometry* folder is the usual *geometry* package, with a few slight changes, and must be on the classpath for all stages of the assignment.

Indicated marks are out of a total of 45. This assignment contributes 6.5% to your final mark.

A. Ray Tracing



The folder *RayTracer* contains a simple ray tracer written for 715. The top level of the package hierarchy for the ray tracer contains classes *RayTracer* and *ViewAndProjectionPanel*. You aren't meant to understand these two classes in any detail. *RayTracer* is the main program, responsible for setting up and handling the rudimentary GUI, shown in the figure on the right. When the *Go (New file)* button is clicked, the user is provided with a standard file browser window, though which to select a scene description file *.wrl. This scene file is defined in a subset of VRML called *miniVRML*. The syntax of *miniVRML* is attached in the Appendix 1 and there are a couple of simple examples included with the assignment. A parser (which you certainly aren't meant to understand) reads that file, constructs from it an object of type *scene.RayTraceableScene*, and calls its *rayTrace* method. All being well, an image of the scene, as seen from the specified view, is drawn in the image panel on the right half of the frame. To draw an image of the same scene file, but with a different view, use the *Go (Old file)* button.

Your job is to fill in some of the missing code as indicated in the steps below. All of the changes you will make should be in the *scene* package or its sub-packages: look through that carefully and make sure you understand it before proceeding. Appendix 2 contains the class diagrams for the *scene* and *scene.illumination* packages, plus an overview of the classes in each package.

(a) Implement the *firstObjectHit* and the *normalAtHit* methods for the *Sphere* class. The *firstObjectHit* method returns either *this* if the given ray hits the sphere, or *null* if it misses. Furthermore, it has to define the *SceneNode* superclass's instance variables *tHit* and *ray* – look carefully at the *SceneNode* and *RayTraceableScene* classes to see why. If the *firstObjectHit* and the *normalAtHit* methods are correctly implemented, the program should then be able to render scenes made from arbitrarily transformed diffuse spheres[1]. The provided files *jellyBeans.wrl* and *sillyHead.wrl* are examples. Obtain a JPEG image of *sillyHead.wrl*, from any suitable viewpoint, called *sillyHead.jpg*.

[7 marks]

(b) Add shadow testing. Appendix 2 explains how I expect shadows to be handled. You'll find that most of the framework is there, but not the shadow testing code itself. When you have shadows working, generate a new version of *sillyHead.jpg* from a viewpoint (0, -2, 10) to show the shadows of the nose and eyes.

[5 marks]

[1] If you find the speed of the ray tracer too painfully slow, you can speed up testing by shrinking the window *before* starting the ray trace. This can give you a speed up factor of four or more. However, this shouldn't be necessary if you have a reasonably fast machine and a modern Java implementation -- on a 750MHz Pentium III with JDK1.3, the *sillyFace* image takes about 5 seconds to compute, including shadows.

(c) Add reflection, again following the design described in Appendix 2. Since VRML does not include any way of specifying mirror reflection, miniVRML has been extended by the addition of a *mirrorColor* field to the *Material* node. Note that you will not be able to view a scene file with “mirrorColor” values in it with a standard VRML viewer. For reflective objects, the normal Phong surface colour will be calculated first and then a reflection ray will be traced. The colour along that ray will be multiplied by *mirrorColor* and added to the accumulated surface colour. Refraction is not required. Generate an image file *reflections.jpg* from the test file *shinyEyeballs.wrl*, using a look-at point of (0, 1.5, 0) and field of view of 15 degrees.

[5 marks]

B. Curves and Surfaces

1. [4 marks] Prove that the cubic Hermite curve is invariant under translation, ie. if $\mathbf{p}_1$ and $\mathbf{p}_4$ are translated by a constant vector $\mathbf{v}$ then every point $\mathbf{p}(t)$ of the original curve is translated by $\mathbf{v}$.

2. [5 marks] Given three control points $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, and $\mathbf{p}_{i+2}$, a quadratic uniform B-spline segment can be defined as the quadratic Bezier curve that has control points $(\mathbf{p}_i+\mathbf{p}_{i+1})/2$, $\mathbf{p}_{i+1}$ and $(\mathbf{p}_{i+1}+\mathbf{p}_{i+2})/2$. Prove from this definition that the equation for a quadratic B-spline can be written as

$$\mathbf{p}(t) = \begin{pmatrix} t^2 & t & 1 \end{pmatrix} \mathbf{M}_{bs} \begin{pmatrix} \mathbf{p}_i \\ \mathbf{p}_{i+1} \\ \mathbf{p}_{i+2} \end{pmatrix}$$

where

$$\mathbf{M}_{bs} = \frac{1}{2}\begin{pmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

[This was part of a question from the 1993 07.415 Final Test]

3. [5 marks] A Catmull-Rom spline curve is a C_1 continuous piecewise cubic with the following properties:

- It directly interpolates (i.e., passes through) all its control points
- Its parametric tangent vector at point $\mathbf{p}_i$ is $(\mathbf{p}_{i+1} - \mathbf{p}_{i-1})/2$.

From this definition, *derive* the Catmull-Rom basis matrix. You should get the answer:

$$\mathbf{M}_{CR} = \frac{1}{2}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix}$$

4. (a) [4 marks] Prove that the quadratic Bezier with the following 2D homogeneous coordinate control points defines a 2D quarter circle: (0,1,1), (√2/2, √2/2, √2/2), (1,0,1). Where is the centre and what is the radius? [Note: this is algebraically tedious if done by hand. A package like *Maple* or *Mathematica* is strongly recommended.]

(b) [1 mark] Using the result from (a), what would be the control points and knots for the equivalent quadratic NURB curve (still in 2D)?

5. [5 marks] The attached extract from Hill (Appendix A4.2) explains how the lid of the famous graphics teapot is defined. Your job is to write a program *TeapotLid.java* that displays a wire-frame mesh of that lid (or, for fewer

marks, a quadrant of it), using *Magician/OpenGL*. You should use the *WireframeHouseWithTrackball* program available from the 715 assignment web page as a startpoint. You must write the code to display the mesh yourself – you are not allowed to use the NURBS-rendering capabilities provided in the OpenGL GLU package. To save you the effort of retyping the vertices from Hill into your program, they are provided for you on the 715 assignment web page as *TeapotLidVertices.java*.

The user should be able to rotate the image using the virtual trackball. There should be a single text field (or slider if you want to be a bit more fancy) in the output window that allows control of the degree of subdivision of the mesh. You should allow a variation from no subdivision (i.e. s and t take the values 0 and 1 only for each quadrant) through to some suitably fine structure.

6. [4 marks] Extend your program *TeapotLid.java* to include a check box that, when checked, causes the teapot to be displayed Gouraud shaded rather than by a wireframe. In order to do this you have to compute the vertex normals. Explain in the written part of your answers to part B how you derived the vertex normals.

Handing In

Hand in via the departmental drop box the following files:

- *Sphere.java*
- Any java source files you modified in part A to implement shadows and reflections.
- The image files *sillyHead.jpg* and *reflections.jpg* (NOTE: The images must have a resolution of at least 300x300 pixels!)
- A text file *ReadMe.txt* (or *ReadMe.doc* or *ReadMe.html*) that indicates what stages you did and includes any other useful information, like derivations of any formulae you used that are not in the notes, known bugs, etc.
- A Word document *Ass2PartB.doc* with the answers to part B of the assignment (ie. Answers to question 1-4 and an explanation of how you derived the vertex normals for question 6). In order to typeset the equations you might want to use the 'Equation Editor' or 'MathType'. **Since the typesetting of mathematical equations is rather tedious you can also write your answers to these questions by hand and hand them in during the lectures.**
- A single file *TeapotLid.java* containing the answers to questions 5 and 6 of part B (with any additional (local) classes defined in that same file) – submit that file via the usual drop box.

Appendix 1. *MiniVRML Syntax*

```
// Grammar for a trivial subset of VRML97.
// Allows only a list of lights followed by a list of scene nodes.
// Lights can be either DirectionalLights [each with an optional
// ambient value (default is 0) and mandatory intensity and
// direction values (in that order) and nothing else] or
// PointLights (each with an optional ambient value and mandatory
// intensity and location values).
// SceneNodes are either TransformNodes or ShapeNodes.
// A TransformNode may have zero or one Translation, Rotation and Scaling
// specifiers (in that order).
// A ShapeNode has a mandatory Appearance and a mandatory Geometry.
// Appearance is a Material with a mandatory diffuseColor and optionally
// both a specularColor and a shininess (for a Phong material).
// An extra mirrorColor field has been added to support reflective ray tracing.
// Geometry is either a box (with mandatory size), an indexed face set,
// a Sphere, a Cylinder or a cone. Spheres, Cylinders and cones have optional radii
// values and cylinders and cones have an optional height. Boxes, spheres and
// cylinders are all centred at the origin.
// An indexed face set has mandatory "coord" and "coordIndex" fields.
VRMLScene ::= Lights SceneNodes
Lights ::= (Light)+
Light ::= DirectionalLight | PointLight
DirectionalLight ::= "DirectionalLight" "{" ["ambient" AmbientValue]
       "intensity" IntensityValue "direction" DirectionValue "}"
PointLight ::= "PointLight" "{" ["ambient" AmbientValue]
       "intensity" IntensityValue "location" LocationValue "}"
AmbientValue ::= FloatValue
IntensityValue ::= FloatValue
DirectionValue ::= Vec3f
AmbientValue ::= FloatValue
IntensityValue ::= FloatValue
DirectionValue ::= Vec3f
LocationValue ::= Vec3f

SceneNodes ::= (SceneNode)*
SceneNode ::= TransformNode | ShapeNode
TransformNode ::= "Transform" "{" [Translation] [Rotation] [Scaling]
       "Children" "[" (SceneNode)* "]" "}"
Translation ::= "translation" Vec3f
Rotation ::= "rotation" Vec4f
Scaling ::= "scale" Vec3f
ShapeNode ::= "Shape" "{" "appearance" AppearanceNode "geometry" GeometryNode "}"
AppearanceNode ::= "Appearance" "{" "material" "Material" "{"
       "diffuseColor" Vec3f ["specularColor" Vec3f "shininess" FloatValue]
       ["mirrorColor" Vec3f] ["transparency" FloatValue] "}" "}"
GeometryNode ::= BoxNode | IndexedFaceSetNode | SphereNode | CylinderNode | ConeNode
BoxNode ::= "Box" "{" "size" Vec3f "}"
IndexedFaceSetNode ::= "IndexedFaceSet" "{" "coord" "Coordinate" "{"
       "point" "[" (FloatValue)* "]" "}"
       "coordIndex" "[" (IntValue)* "]" "}"
SphereNode ::= "Sphere" "{" ["radius" FloatValue] "}"
CylinderNode ::= "Cylinder" "{" ["radius" FloatValue] ["height" FloatValue] "}"
ConeNode ::= "Cone" "{" ["bottomRadius" FloatValue] ["height" FloatValue] "}"
Vec3f ::= FloatValue FloatValue FloatValue
Vec4f ::= FloatValue FloatValue FloatValue FloatValue
FloatValue ::= <SFFloat>
IntValue ::= <SFInt>
```

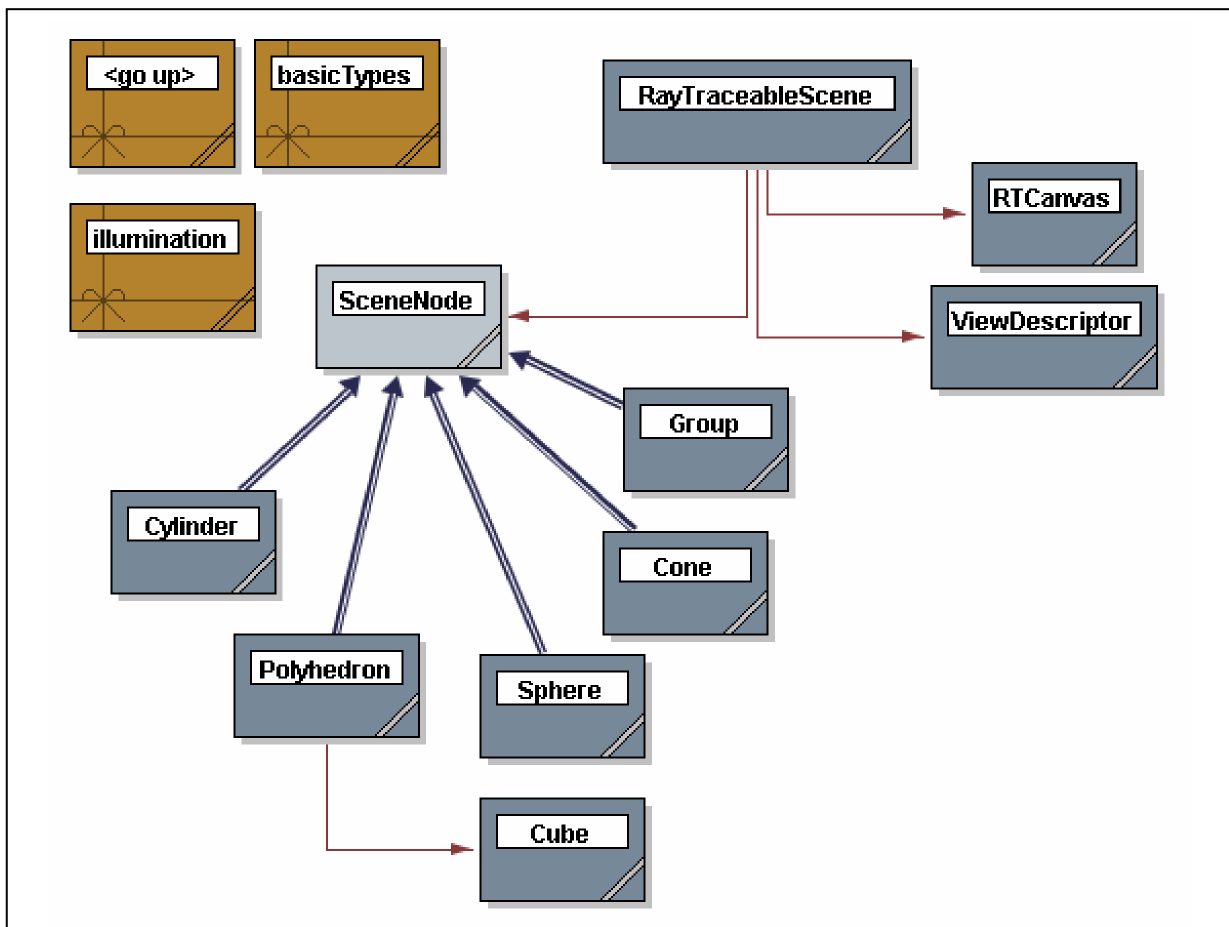
Appendix 2. Notes on the Design of the Ray Tracer

The scene package

The figure below shows the design of the *scene* package, as generated by the *blueJ* IDE (http://www.bluej.org). The *scene* package uses sub-packages *basicTypes*, which contains just the *Colour* and the *Ray* classes, and *illumination*, which will be discussed shortly. A *RayTraceableScene* contains a scene graph and a vector of lights. Its *rayTrace* method is called to render the scene onto an *RTCanvas*, which is just a canvas with a *setPixel* method. The *ViewDescriptor* class defines the eye point, field of view and view up vector, and also contains a method to return the *Ray* from the eye through a given pixel on the image plane.

The scene graph is defined by a single object of type *SceneNode*, which is the root of the scene graph (actually a tree) and will usually be of the *Group* subclass. A *Group* object is a vector of *SceneNodes*, some of which may themselves be *Group* nodes so that an n-ary tree structure is possible. The other subclasses of *SceneNode* are the various leaf nodes of the scene tree, each representing a particular geometric shape.

Ray tracing a scene tree is performed by calling the *firstObjectHit(ray)* method on the root of the tree. This is defined to return the leaf node that is the first object hit by the given ray, or null if the ray misses the scene graph. Other information relating to the ray-object intersection, such as the distance along the ray, is cached within the leaf node, and can be queried by subsequent method calls to the returned object.



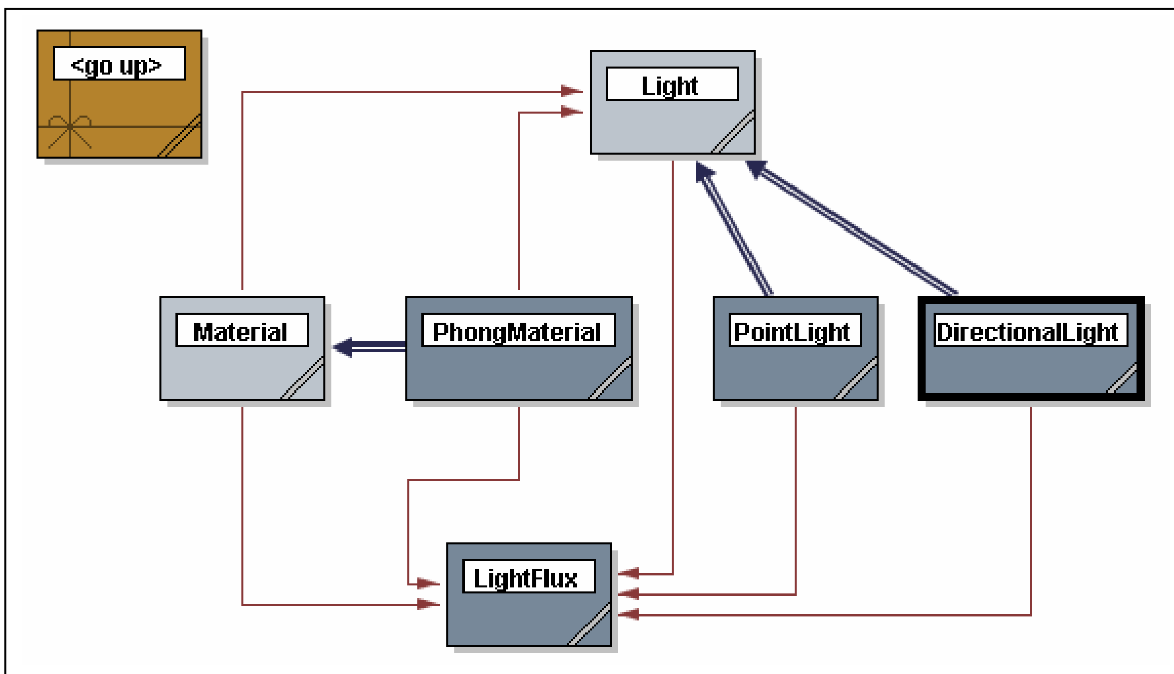
The scene.illumination package

The figure below shows the design of the *illumination* sub-package of the *scene* package. This contains classes related to illumination and surface reflectance properties. An instance of class *Material* defines the surface reflectance properties of a leaf node in the scene graph. Essentially it just provides a method to return the colour of the material given a vector of lights, a point on the surface, the surface normal at that point, and the direction vector from the surface point to the eye. The subclass *PhongMaterial* implements the standard (sort of) reflection model, but with the addition of a "mirror colour" parameter, which is used to implement mirror reflections.

A *Light* object is something that provides illumination. The illumination provided by a light is represented by a single *LightFlux* object, which is just a record containing both a light ambient component and a directional component. The *PointLight* and *DirectionalLight* subclasses of *Light* implement specific types of light.

Implementation of shadows proves a little tricky in this OO framework. The solution used is a little unorthodox: the light returns a light flux object with a null directional component if the light source can't actually see the point at which the illumination is required. This fits naturally into the definition of the light's *illumination* method, but does create a little inelegance in implementation. The light needs to know the complete scene it is illuminating in order to determine whether or not it can "see" the point of interest. Hence, the *Light* class includes a *setScene* method to tell the light the root of the scene tree; this method is called by the constructor of the *RayTraceableScene*. When calculating the light flux at the point of interest, the light has to cast a ray towards the point of interest and determine if there is anything in the way.

Implementation of reflection follows the same general design approach as shadows, with the same sort of inelegancies. In order to return the colour at a point, the *Material* needs to reflect the vector to the eye about the surface normal, and cast a ray back into the scene. This means telling all materials the entire scene – scene graph plus lights. So the *Material* class has a *setScene* method that takes a *RayTraceableScene* as a parameter and the method just records that value. The constructor for a *RayTraceableScene* ensures that method is called on all materials in the scene by using the *setSceneOnAllMaterials* method of a *SceneNode.*



Enjoy!