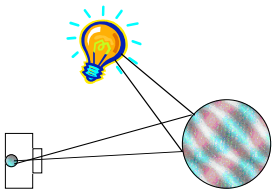
UPPSALA UNIVERSITET

Methods of Programming DV2

Introduction to ray tracing and XML

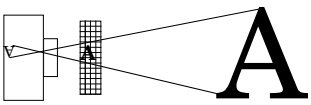
Ray tracing

Suppose we have a description of a 3-dimensional world consisting of various objects – spheres, triangles (flat), planes (flat, infinite) and point light sources.
Construct the 2-dimensional image created by a camera (or eye), by simulating the light rays emitted by the light sources.



The image plane

To avoid the complications of the lens and inverted image, in the computer model, the lens is assumed to be a point (cf. pinhole cameras) and the light rays entering the camera are captured in an imaginary *image plane* located in front of the camera.



The image plane consists of a number of pixels (picture elements – square areas with a single colour). The light rays passing each pixel on their way to the camera determine the picture.

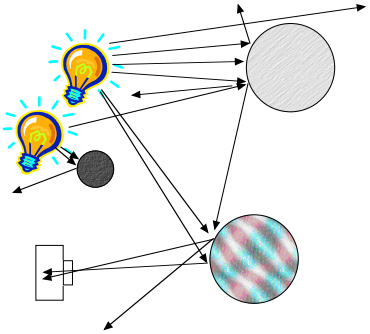
Forward ray tracing

- Follow light rays from each light source!
- When a light ray strikes a surface, determine how the light ray is:
  - Reflected
  - Coloured
  - Absorbed

  depending on surface properties.
- Follow all reflected rays recursively until we find a ray that passes the image plane.
- For each pixel in the image plane, accumulate the colour and intensity of each ray. In the end, we will get the complete picture.

Problem: There are very (infinitely…) many light rays emitted and reflected in every possible direction. Most of them will never pass through the image plane. Computationally infeasible.

Forward ray tracing (example)

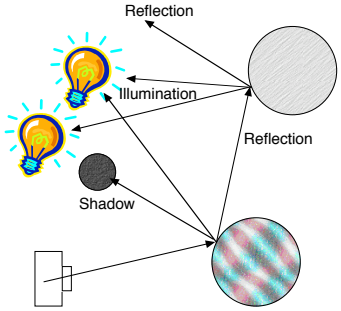


...just some of the rays emitted from the light sources…

Backward ray tracing

- Instead, follow light rays *backward* from the camera.
- Only one ray for each pixel.
- When (if) the backward ray hits an object, determine the intensity of light and colour coming from the object at the *intersection point*.
- If the object is a light source, this is straightforward. Otherwise:
  - Recursively trace a new backward ray *reflected* by the surface.
  - Recursively trace a new backward ray from the intersection point to *each* light source on the ”strike side” of the surface. That light source will contribute to the *illumination* of the intersection point.
  - (Possibly handle light *refracted* in transparent surfaces similarly.)
- The amount of light reaching the pixel depends on the angle of the ray to the surface, as well as absorption properties of the surface. The light colour may also be affected by the surface colour.

Backward ray tracing (example)



One ray traced backwards from a particular pixel in the image plane.

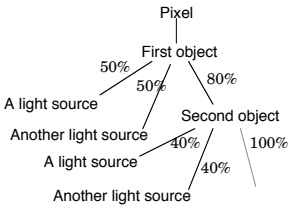
When to stop?

- As the image plane has a fixed number of pixels, we have a limited number of rays to start with.
- A ray can be reflected any number of times. When do we stop following the ray?

- Two approaches:
  - We stop after a set number of reflections.
  - At every reflection, the light ray is weakened by absorption in the surface. We stop when the accumulated absorption becomes great enough.

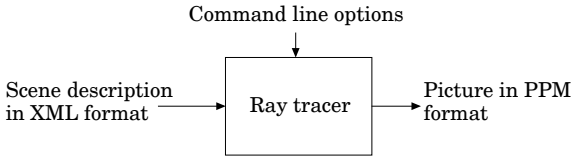
Ray trees

- The tracing of the ray for a particular pixel in the image plane can be described by a *ray tree*.



- The percentages tell how much of the light reaching the object along the branch is transmitted (not absorbed).
- The tree shows the structure of the computation and need not actually be constructed.

Ray tracer input and output



- XML is described in this lecture.
- PPM is described in the project specification.
- Command line options (things like image plane parameters, recursion limit…) are described in the project specification.

What is XML?

XML is a standard way of representing *structured* data in a way which is

- Portable (system-independent)
- Human readable (at least to some extent)
- Machine readable
- Easy to create and process.

XML (Extensible Markup Language, W3C recommendation 1998) is similar in concept and appearence to HTML (Hypertext Markup Language, IETF RFC 1995), as they both derive from SGML (Standard Generalised Markup Language, ISO standard 1986). Indeed, modern versions of HTML (XHTML) are based on XML. XML describes the *structure*, not the *meaning*, of data using *markup*.

Sample XML file

Ray tracer input data (scene descriptions) are stored in an XML file such as this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE scene SYSTEM
      "http://www.it.uu.se/edu/course/homepage/pm2/vt09/pm2gl.dtd">
<!--  This is a comment in XML -->
<scene>
  <camera>
    <location>
      <vector x="0.000" y="3.000" z="-10.000"/>
    </location>
    <sky>
      <vector x="0.000" y="1.000" z="0.000"/>
    </sky>
    <look_at>
      <vector x="0.000" y="0.000" z="0.000"/>
    </look_at>
  </camera>
  <world>
    ……
  </world>
</scene>
```

XML prologue

XML declaration:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Identifies the file as an XML document, provides XML version and character encoding.

Document Type Declaration (DTD):

```
<!DOCTYPE scene SYSTEM
      "http://www.it.uu.se/edu/course/homepage/pm2/vt09/pm2gl.dtd">
```

Specifies the main element in the file and defines the structure of the file — valid elements and how they may be combined.

In this case the details (declarations) are stored in a DTD file with the given URI (Uniform Resource Identifier – ”web address”).

The DTD file can be read by programs processing XML files (e.g. editors) to provide error checking.

XML data

Data is structured using nested *elements*.

`scene` is a *container element* with contents delineated by a start and end *tag*.

There may be ordinary text data between tags in container elements, but this is not used in XML files for our ray tracing data.

`vector` is an *empty element*, which however has *attributes* (`x`,`y`,`z`).

Container elements can also have attributes.

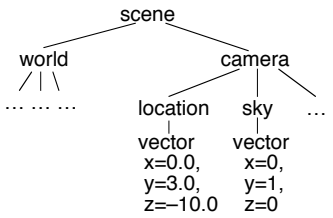
Empty element tags have to close with ”/>”.

```
<camera>
  <location>
    <vector x="0.000" y="3.000" z="-10.000"/>
  </location>
  <sky>
    <vector x="0.000" y="1.000" z="0.000"/>
  </sky>
  ……
</camera>
```


XML file structure

The XML elements define a tree structure on the data:

```
<scene>
  <camera>
    <location>
      <vector x="0.000" y="3.000" z="-10.000"/>
    </location>
    <sky>
      <vector x="0.000" y="1.000" z="0.000"/>
    </sky>
    ……
  </camera>
  <world>
    ……
  </world>
</scene>
```



DTD element declararations

`<!ELEMENT` *`name contents`*`>`

This declares that *name* is a valid element name and that each element of that name must have contents according to *contents*.

contents can be one of:

`element`	An element with a particular name.
`contents``?`	Optional contents.
`contents``+`	Repeated contents (at least once).
`contents``*`	Optional repeated contents.
(*`contents1`*`,`*`contents2`*`,…)`	Several pieces of content in the given order
(*`contents1`*`|`*`contents2`*`|…)`	Either one of the pieces of content.
`#PCDATA`	Character data outside tags.
`EMPTY`	No contents allowed.

Example:

```
<!ELEMENT scene (camera, background?, world)>
```


DTD attribute declararations

`<!ATTLIST` *`element`*
`attname type optional`
…possibly more attributes…`>`

This declares that *`attname`* is a valid attribute of *`element`*.

`optional` can be one of

`#REQUIRED`	The attribute must be present.
`#IMPLIED`	The attribute is optional.

`type` can be one of:

`CDATA`	Quoted character data.
(*`keyword1`*`|`*`keyword2`*`|..)`	One of the keywords.

Example:

```
<!ATTLIST vector
  x CDATA #REQUIRED
  y CDATA #REQUIRED
  z CDATA #REQUIRED>
```


The ray tracer DTD – worlds, scenes, colours

A scene consists of a camera, a world, and a possible background.

```
<!ELEMENT scene (camera, background?, world)>
```

A world consists of a objects and lights sources.

```
<!ELEMENT world (plane | light | sphere | triangle)*>
```

A background has a (uniform) colour.

```
<!ELEMENT background (color)>
```

A colour has no contents, but does have RGB components.

```
<!ELEMENT color EMPTY>
<!ATTLIST color
   red CDATA #REQUIRED
   green CDATA #REQUIRED
   blue CDATA #REQUIRED>
```

The components are character representations of floating point numbers between 0 and 1. This can not be enforced by the DTD but must be verified by the ray tracer.

The ray tracer DTD lights, positions, vectors

Lights sources are points at a given position in the 3D space.

```
<!ELEMENT light (position)>
```

A 3D position is a vector from the coordinate system origin.

Sometimes the element `location` is used for the same purpose.

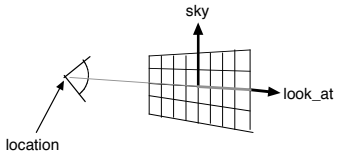
```
<!ELEMENT position (vector)>
<!ELEMENT location (vector)>
```

A vector has no contents, but does have x, y and z coordinates..

```
<!ELEMENT vector EMPTY>
<!ATTLIST vector
  x CDATA #REQUIRED
  y CDATA #REQUIRED
  z CDATA #REQUIRED>
```

The coordinates are character representations of floating point numbers. This can not be enforced by the DTD but must be verified by the ray tracer.

The ray tracer DTD – the camera



The camera is defined by its location and the orientation in space of its image plane.

```
<!ELEMENT camera (location, sky, look_at)>
<!ELEMENT sky (vector)>
<!ELEMENT look_at (vector)>
```

Note that the size of the image plane as well as its number of pixels and distance from the camera (”zoom value”) is not in the XML file. That information is given as parameters to the ray tracer.

The ray tracer DTD – surfaces

```
<!ELEMENT surface (finish?, pigment?)>
```

A surface can have an optional pigmentation – either a uniform colour or a picture taken from a file.

```
<!ELEMENT pigment (color|image)>
<!ELEMENT image (ppm)>
<!ELEMENT ppm EMPTY>
<!ATTLIST ppm file CDATA #REQUIRED>
```

The `file` attribute should be the name (path) of a ppm picture file.

A surface can have an optional ”finish” which tells how it affects light that falls on it. It can optionally reflect and/or diffuse light

```
<!ELEMENT finish EMPTY>
<!ATTLIST finish
   diffuse CDATA #IMPLIED
   reflect CDATA #IMPLIED>
```

The diffusibility and reflexivity are character representations of floating point numbers between 0 and 1. This can not be enforced by the DTD but must be verified by the ray tracer.

The ray tracer DTD – triangles

A triangle is defined by its corners or by one corner and two sides. It can have an optional surface description.

```
<!ELEMENT triangle (c0, ((v1, v2) | (c1, c2)), surface?)>
```

`<!ELEMENT c0 (vector)>`	First corner
`<!ELEMENT c1 (vector)>`	Second corner
`<!ELEMENT c2 (vector)>`	Third corner
`<!ELEMENT v1 (vector)>`	First side
`<!ELEMENT v2 (vector)>`	Second side

The ray tracer DTD – planes

A plane is defined by a vector at right angle to it (a normal) and either one point on the plane or the distance of the plane from the coordinate system origin. It can have an optional surface description

```
<!ELEMENT plane (normal, point?, surface?)>
<!ELEMENT normal (vector)>
<!ELEMENT point (vector)>
<!ATTLIST plane distance CDATA #IMPLIED>
```

The distance is a character representation of a floating point numbers. This can not be enforced by the DTD but must be verified by the ray tracer.

The DTD can not express that either a point or a distance, but not both, is given. This has to be verified by the ray tracer. (If none is given, a zero distance should be assumed.)

The ray tracer DTD – spheres

A sphere is defined by its location and radius. It can have an optional surface description.

```
<!ELEMENT sphere (location, pole?, equator?, surface?)>
<!ATTLIST sphere radius CDATA #REQUIRED>
```

The radius is a character representation of a floating point number. This can not be enforced by the DTD but must be verified by the ray tracer.

In case the surface is a picture, the pole and equator vectors tell how the picture should be oriented on the sphere (see the project specification).

```
<!ELEMENT pole (vector)>
<!ELEMENT equator (vector)>
```

Sample XML files

A number of test files can be found in the department Unix directory `/it/kurs/pm2/vt09/xml`.

See the course web site for info on the xmlpro editor which lets you create, view or modify XML files according to the tree structure of the file.

Information on how to use the libxml2 XML parser library to read XML files from your ray tracer program will be given later in the course.