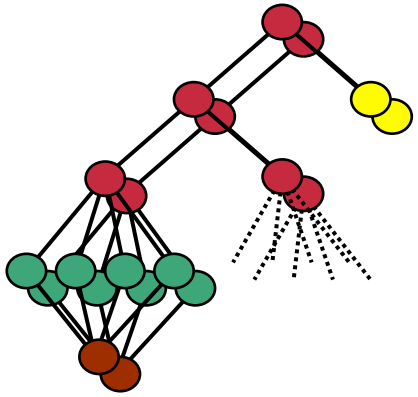
Bremen

CG VR

Virtual Reality & Physically-Based Simulation
Scenegraphs & Game Engines

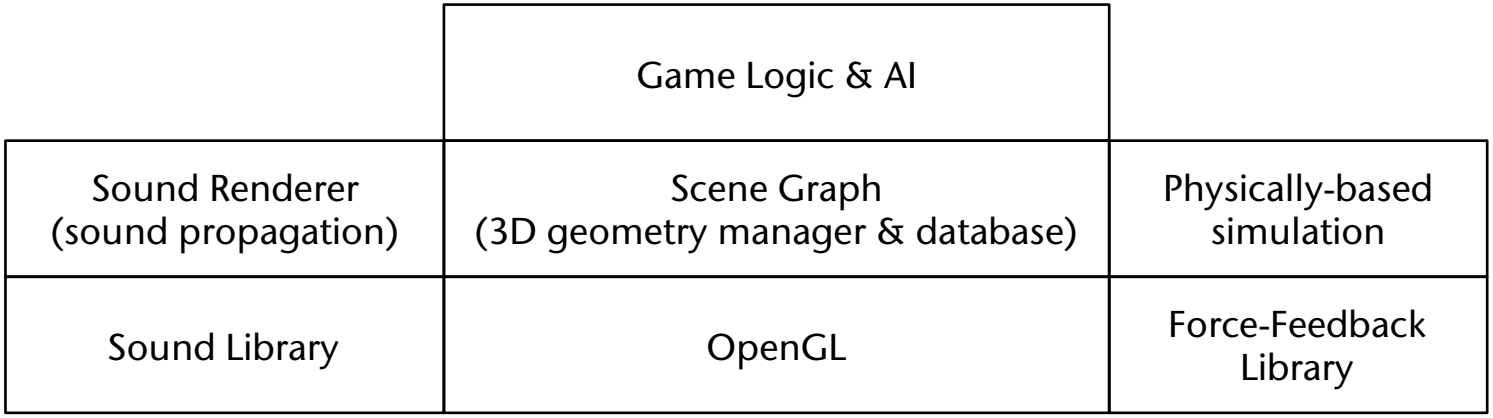


G. Zachmann

University of Bremen, Germany

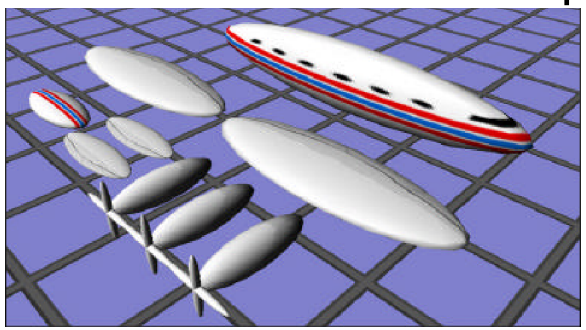
cgvr.cs.uni-bremen.de

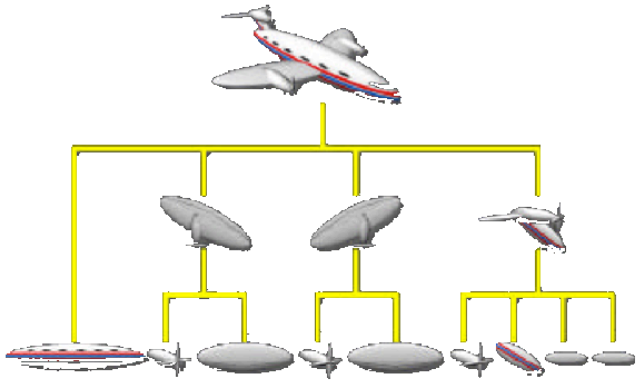
Overall System Architecture



Motivation

- Immediate mode vs. retained mode:
  - Immediate mode = OpenGL / Direc3D = Application sends polygons / state change commands to the GPU → flexibler
  - Retained mode = scenegraph = applications builds pre-defined data structures that store polygons and state changes → easier and (probably) more efficient rendering
- Flat vs. Hierarchical scene descriptions:

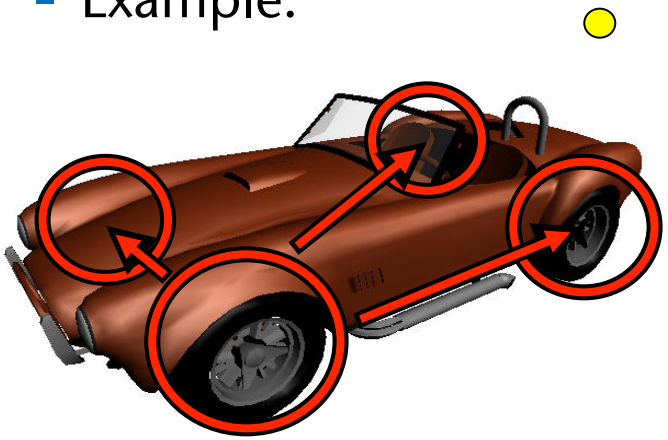


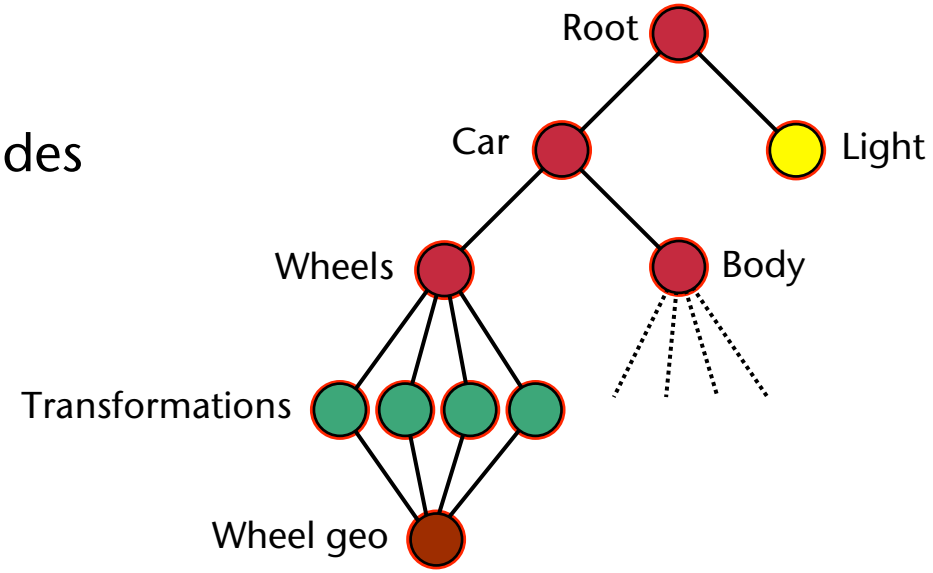


- *Code re-use* and *knowledge re-use*!
- Descriptive vs. imperative (cv. Prolog vs. C)
  - Thinking objects ... not rendering processes

Structure of a Scene Graph

- Directed, acyclic graph (DAG)
  - Often even a proper tree
- Consists of heterogeneous nodes
- Example:





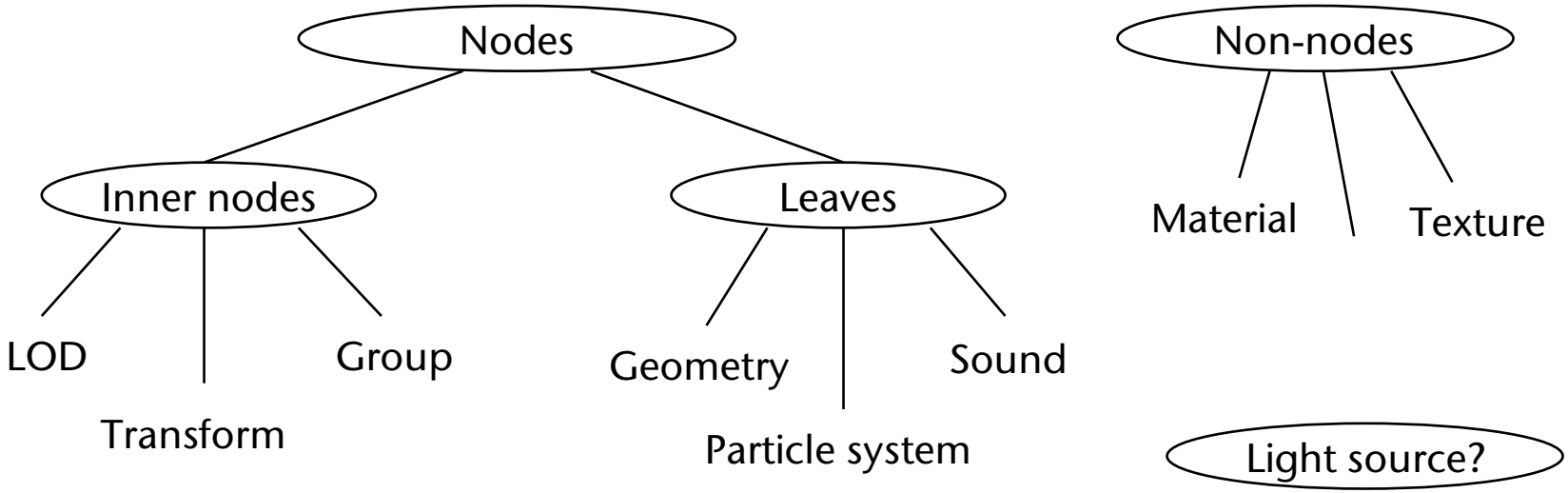
- Most frequent operation on scene graph: rendering
  - Amounts to depth-first traversal
  - Operation per node depends on kind of node

Semantics

- Semantics of a node:
  - Root = "universe"
  - Leaves = "content" (geometry, sound, ...)
  - Inner nodes = forming groups, store state (changes), and other non-geom. functionality, e.g., transforms
- Grouping: exactly how is left to the application, e.g., by
  - Geometric proximity → scenegraph induces a nice BVH
  - Material → good, because state changes cost performance!
  - Meaning of nodes, e.g., all electrical objs in the car under one node → good for quickly switching off/on all electrical parts in the car
- Semantics of edges = inheritance of states
  - Transformation
  - Material
  - Light sources (?)

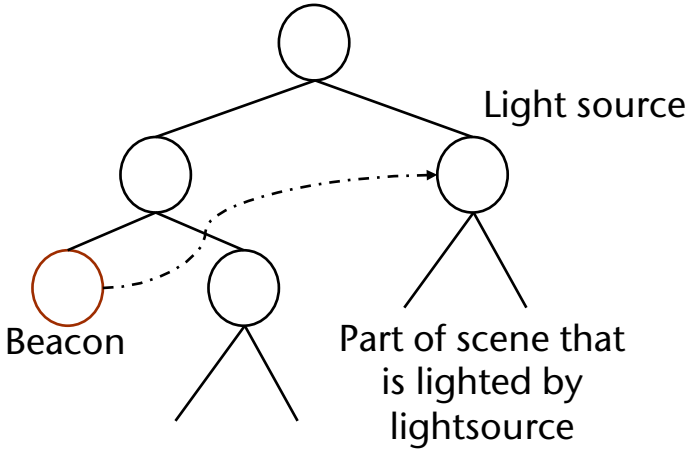
Kinds of Nodes

- There are 2 hierarchies: scenegraph hierarchy + class hierarchy
- The flexibility and the expressiveness of a scenegraph depends heavily on the kinds and number of node classes!
- Some classes (or, rather, their instances) will not be part of the scenegraph, but still be in the overall scene

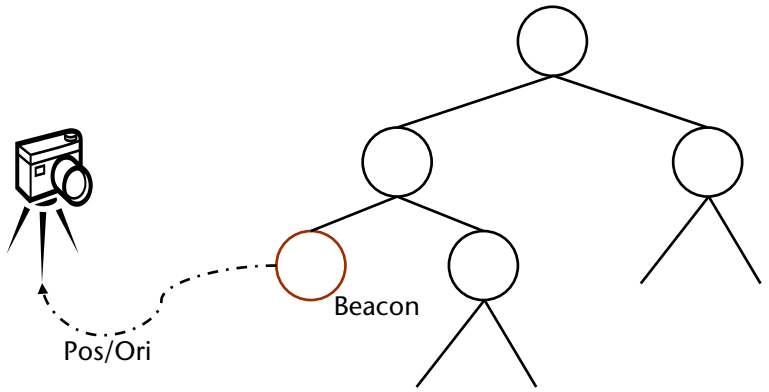


Issues

- Light sources:
  - Usually part of the scenegraph
  - Problem with naïve semantics: what if light source should move/turn, but not the scene it shines on?
  - Solution: beacons
    - Lightsource node lights its sub-scene underneath
    - Position/orientation is taken from the beacon



- Camera: to be, or not to be a node in the scenegraph?
  - Both ways have dis-/advantages
  - If not a node: use beacon principle

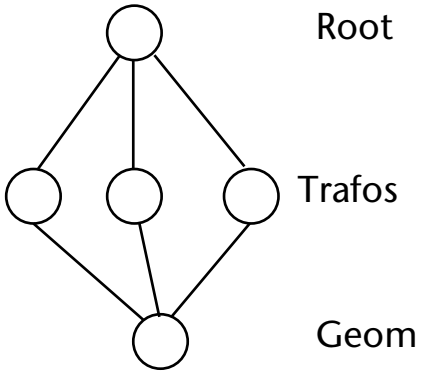


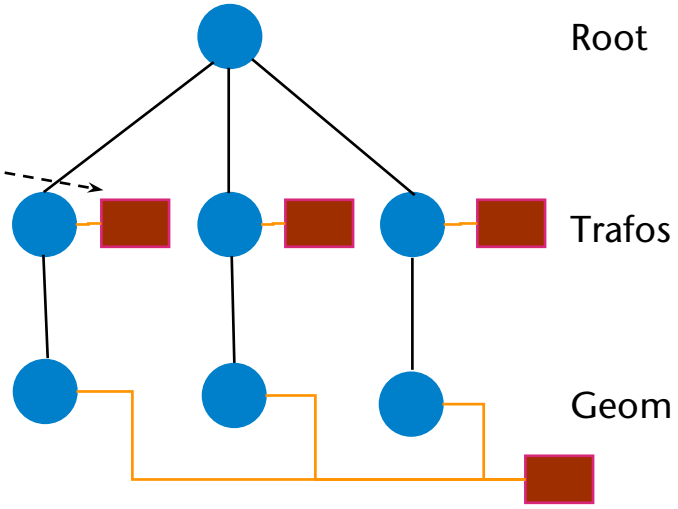
Material

- Material =
  - Color, texture, lighting parameters (see Phong)
  - Property of a node
- Semantics of materials stored with inner nodes: top-down inheritance
  - Path from leaf to root should have at least one material
  - Consequence:
    - Each leaf gets rendered with a unique, unambiguously defined material
    - It's easy to determine it
- Bad idea (Inventor): inheritance of material from left to right!

Sharing of Geometry / Instancing

- Problem: large scenes with lots of identical geometry
- Idea: use a DAG (instead of tree)
  - Problem: pointers/names of nodes are no longer *unique/unambiguous*!
- Solution: separate structure from content
  - The tree proper now only consists of *one kind* of node
  - Nodes acquire specific properties/content by attachments / properties
  - Advantages
    - Everything can be shared now
    - Many scenegraphs can be defined over the same content
    - All nodes can acquire lots of different properties/content





Thread-Safe Scenegraphs for Multi-Threading

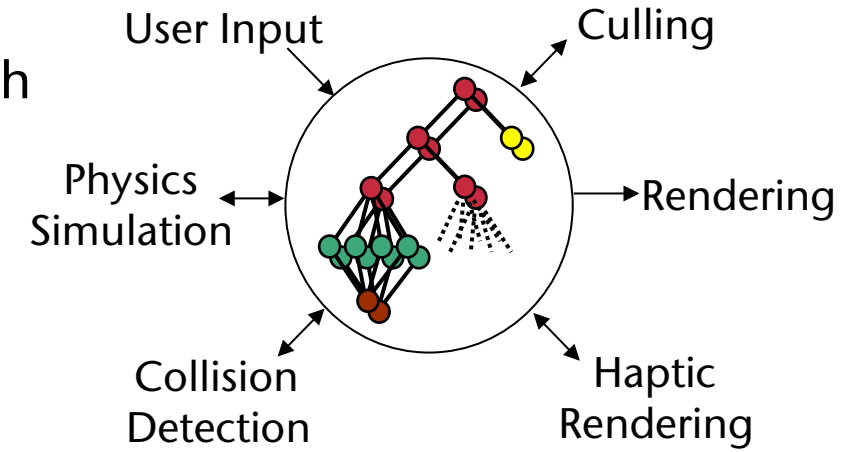
- Idea: several copies of the scenegraph
- Problem: memory usage & sync!
- Solution:
  - *Copy-on-Write* of the attachments
    → "Aspects"
  - Each thread "sees" their own aspect
  - Problem: easy access via pointers

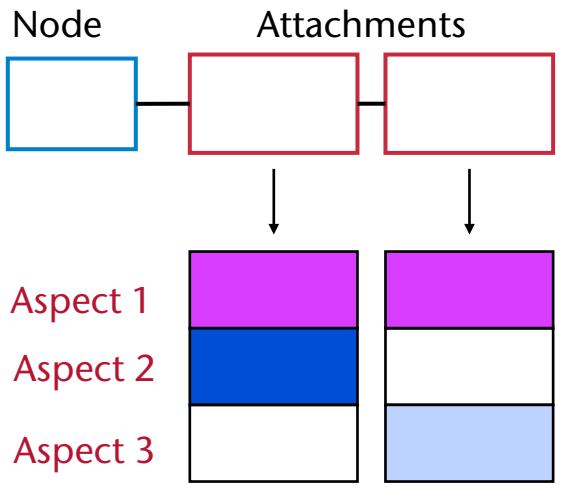
    ```
    geom->vertex[0]
    ```

    does not work any more
  - Solution (leveraging C++):
    - "Smart Pointers"

    ➢Needs one "pointer class" per node. Ex.:

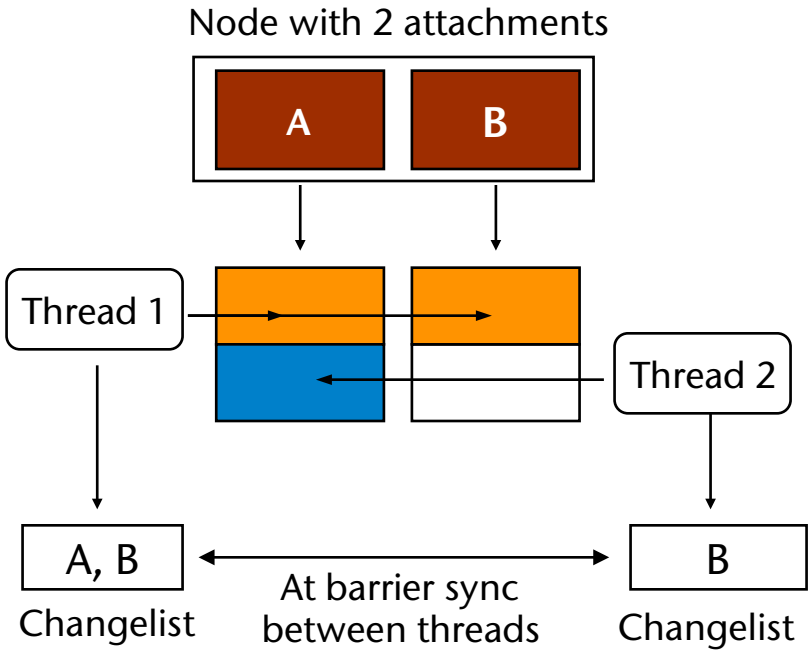
    ```
    geomptr = Geometry::create(…);
    geomptr->vertex[0] ...
    ```





Distributed Scenegraphs

- Synchronisation by changelists
  - Make scene graph consistent at one specific point during each cycle of each thread
    ⟶ barrier synchronization



- Distributed rendering:
  - Goal: rendering on a cluster
  - Problem: changes in the scenegraph need to be propagated
  - Solution: simply communicate the changelists
    - Items in the changelist = IDs of nodes/attachments to be changed + new data

Issue: Memory-Layout for Fast Rendering

Optional

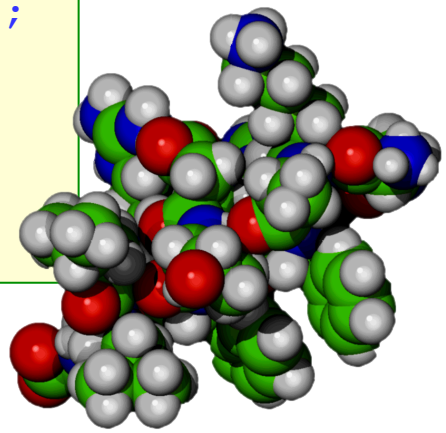
- Frequent problem: the elegant way to structure data (from the perspective of software engineering) is inefficient for fast rendering
- Terminology: "Array of Structs (AoS)" vs. "Struct of Arrays (SoA)"
- For illustration: example of visualization of molecules
  - Following good SE practice, we should design classes like this

```
class Atom
{
public:
    Atom( uint atom_number, Vec3 position, ... );
private:
    Vec3   position_;
    uint   atom_number_;
    Atom * bonds_[max_num_bonds];
    ...
};
```

Optional

- And the class for a molecule:

```
class Molecule
{
public:
    Molecule( const std::vector<Atom> & atoms );
private:
    std::vector<Atom> atoms_;
    ...
};
```



- Memory layout of a molecule is now an AoS:

pos	num	bonds	pos	num	bonds	pos	num	bonds	

Optional

- Problem with that: memory transfer becomes slow
- Alternative: Struct of Arrays

```
class Molecule
{
private:
    std::vector<Vec3>  position;
    std::vector<uint>  atom_number;
   ...
};
```

pos[0] pos[1] pos[2] . . .	atom_number[0] . . .	

Criteria for the Usage of Scenegraphs

- When is a hierarchical organization of the VE effective:
  - Complex scenes: many hierarchies of transformations, lots of different matierals, large environment with lots of geometry of which usually only a part can be seen (culling)
  - Mostly static geometry (opportunities for rendering optimization, e.g., LoD's)
  - Specific features of the scenegraph, e.g., particles, clustering, ...
- When not to use a hierarchical organization / scenegraph:
  - Simple scenes (e.g., one object at the center, e.g., molecular vis)
  - Visualization of scientific data (e.g., CT/MRI, or FEM)
  - Highly dynamic geometry (e.g., all objects are deformable)

Fields & Routes Concept by Way of X3D/VRML

- What is X3D/VRML:
  - Specification of nodes, each of which has a specific functionality
  - Scene-graph definition & file format, plus ...
  - Multimedia-Support
  - Hyperlinks
  - Behavior and animation
  - "VRML" = "Virtual Reality Modeling Language"
- X3D = successor & superset of VRML
  - Based on XML
- VRML = different encoding, but same specification
  - Encoding = "way to write nodes (and routes) in a file"

Hello World

- In X3D (strictly speaking: "XML encoding"):

Like the <html> tag in HTML →
Root node →
Definition of nodes →

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D profile='Immersive'>
<Scene>
  <Shape>
    <Text string="Hello" "world!" />
  </Shape>
</Scene>
</X3D>
```

- In VRML:

No explicit root node in VRML →

```
#X3D V3.1 utf8
Shape {
  geometry Text {
    string [ "Hello" "world!" ]
  }
}
```

Tip: Use an ASCII editor wich identifies *matching brackets* as a text unit

Nodes and Fields

- Nodes are used for describing ...
  - ... the scenengraph (the usual suspects):
    - Geometry, Transform, Group, Lights, LODs, ...
  - ... the behavior graph, which implements all response to user input (later)
- Node := set of fields
  - "Single-valued fields" and "multiple-valued fields"
  - Each field of a node has a unique identifier
  - These are predefined by the X3D/VRML specification
- Field types:
  - field = actual data in the external file
  - eventIn, eventOut = used only for connecting nodes, data that won't be saved in a file
  - exposedField = combination of these (xxx, set_xxx, xxx_changed)

Types of Fields

- All field types exist as "single valued" (**SF**...) and as "multiple valued" kind (**MF**...)
- Example of an SF field:

```
<Material diffuseColor="0.1 0.5 1" />
```

X3D

```
material Material {
  diffuseColor  0.1 0.5 1
}
```

VRML

- MF fields are practically the same as arrays
  - Special notation for signifying an MF field and to separate elements

Optional

- Primitive data types: the usual suspects

Field type	X3D example	VRML example
SFBool	true / false	TRUE / FALSE
SFInt32	12	-17
SFFloat	1.2	-1.7
SFDouble	3.1415926535	
SFString	"hello"	"world"

Reminder: for each SF-field there exists an MF-field

- Higher data types:

Field type	example
SFColor	0 0.5 1.0
SFColorRGBA	0 0.5 1.0 0.75
SFVec3f	1.2 3.4 5.6
SFMatrix3f	1 0 0 0 1 0 0 0 1
SFString	"hello"

Optional

- Special field types:

Field type	X3D example	VRML example
SFNode	**<Shape> ... </Shape>**	**Shape { ... }**
MFNode	**<Shape>**... **, <Group>**... oder **<Transform>**...	**Transform { children [...] }**
SFRotation	**0 1 0 3.1415**	
SFTime	**0**	

Optional

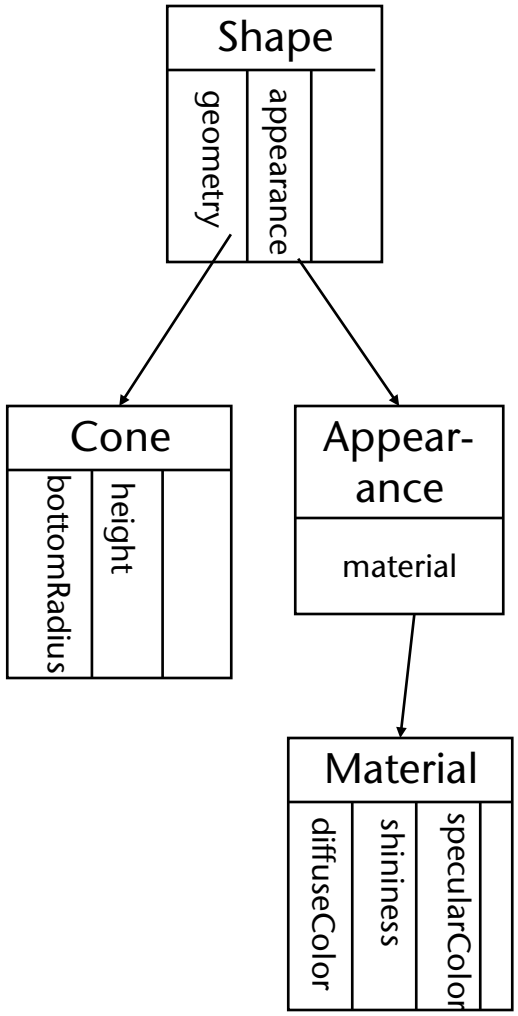
- General remarks on the design of X3D/VRML:
  - The design is orthogonal in that there exists a **`MF`**-type for every **`SF`**-type
  - The design is not orthogonal in that some types are generic (e.g. **`SFBool`**, **`SFVec3f`**) while others have very specific semantics (e.g. **`SFColor`**, **`SFTime`**, etc.)
    - It is not clear whether this is good or bad ...

Nodes to Describe the Scenengraph (Geometry)

- All scenegraphs have a set of different kinds of nodes to define the tree:
  1. Nodes for grouping / hierarchy building
  2. Nodes for storing actual geometry
  3. Nodes for storing appearance, i.e., material def's, textures, etc.
- In X3D/VRML, for instance:
  1. **`Shape, Group, Transform, Switch , Billboard, LOD, ...`**
  2. **`TriangleSet, IndexedTriangleSet, IndexedFaceSet, IndexedTriangleStripSet, Box, Sphere, Cylinder, NurbsPatchSurface, ElevationGrid , ... ...`**
  3. **`Appearance, Material , ImageTexture ,`**

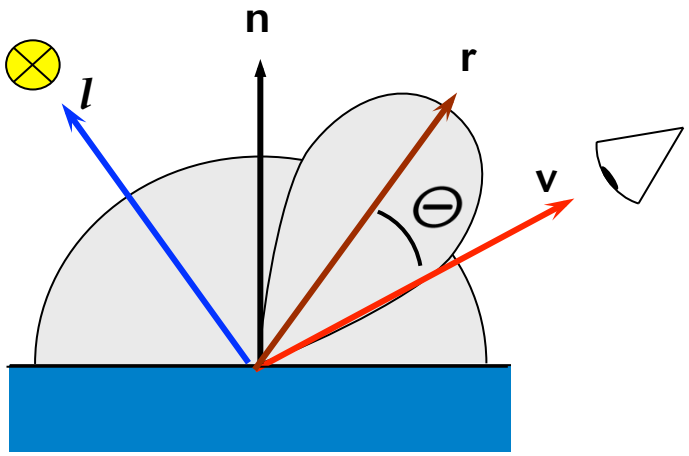
A Simple Example

```
#X3D V3.1 utf8
Shape {
  geometry Cone {
    bottomRadius 1
    height       2
  }
  appearance Appearance {
    material Material {
      ambientIntensity 0.256
      diffuseColor     0.029 0.026 0.027
      shininess        0.061
      specularColor    0.964 0.642 0.980
    }
  }
}
```



Specifying the Material

- Usually, the Phong model is assumed:



$$I_{\text{out}} = I_{\text{amb}} + I_{\text{diff}} + I_{\text{spec}}$$

$$I_{\text{diff}} = I_{\text{in}} \cos\phi$$

$$I_{\text{spec}} = I_{\text{in}} (\cos\theta)^p$$

$$I_{\text{out}} = k_d \cdot I_a + \sum_{j=1}^{n} (k_d \cos \Phi_j + k_s \cos^p \Theta_j) \cdot I_j$$

$$= k_d I_a + \sum_{j=1}^{n} (k_d (\mathbf{n}\boldsymbol{l}) + k_s (\mathbf{r}\mathbf{v})^p) \cdot I_j$$

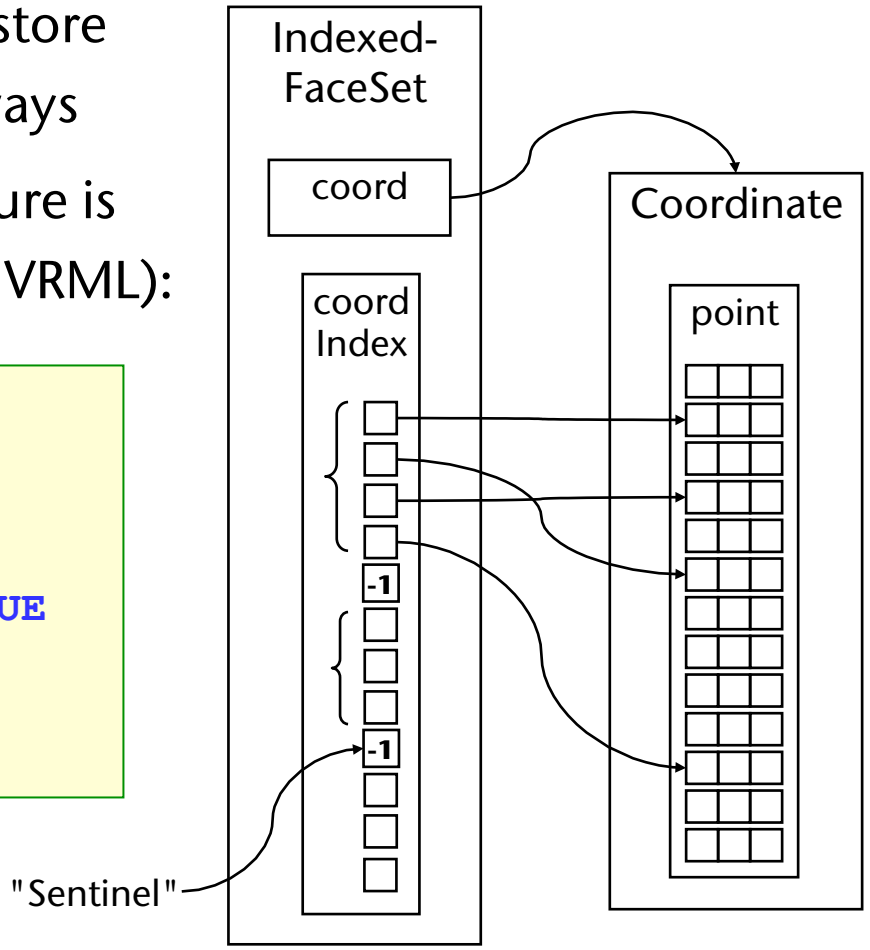
- k_d = diffuse reflection coefficient
  k_s = specular reflection coefficient
  p = *shininess*

- In VRML/X3D:

```
Material {
   SFFloat ambientIntensity 0.2
   SFColor diffuseColor     0.8 0.8 0.8
   SFColor specularColor    0 0 0
   SFFloat shininess        0.2
   SFColor emissiveColor    0 0 0
   SFFloat transparency     0
}
```

Common Data Structures to Specify Geometry

- Most scene graphs and game engines have internal data structures to store geometry in memory-efficient ways
- One very prominent data structure is the **`IndexedFaceSet`** (here in VRML):

```
IndexedFaceSet {
  SFNode  coord          NULL
  MFInt32 coordIndex     []
  SFBool  ccw            TRUE
  SFBool  normalPerVertex TRUE
  SFBool  solid          TRUE
  SFFloat creaseAngle    0.0
}
```

```
Coordinate {
  MFVec3f point []
}
```