SNS COLLEGE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTION)
Approved by AICTE & Affiliated to Anna University
Accredited by NBA & Accredited by NAAC with 'A++' Grade,
Recognized by UGC saravanampatti (post), Coimbatore-641035.

Department of Biomedical Engineering

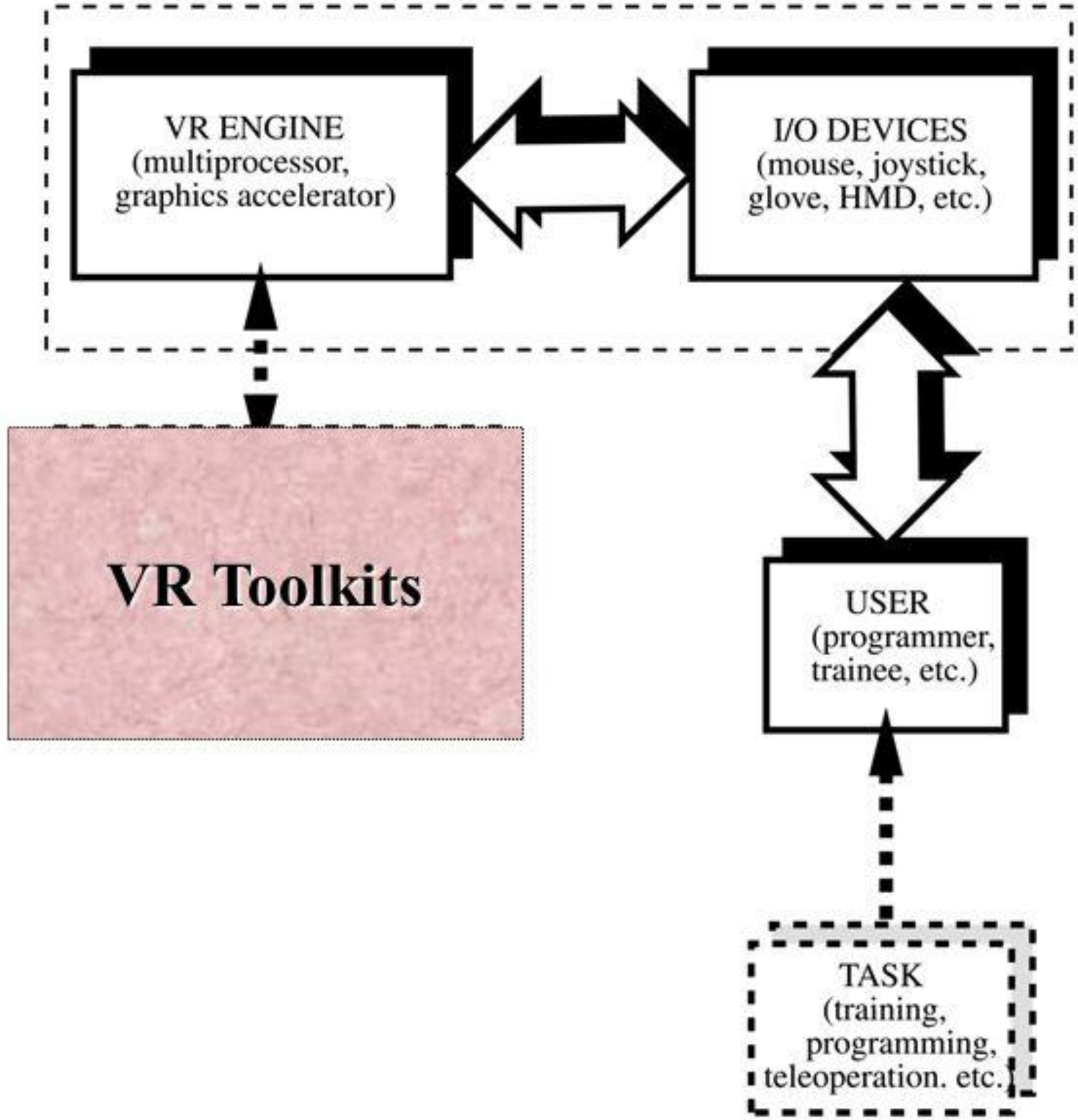
Course Name: 19BMT401 – Virtual Reality in Medicine

IV Year : VII Semester

Unit IV – VR programming

VR PROGRAMMING

VR SYSTEM ARCHITECTURE



System architecture

VR Programming Toolkits

- Are extensible libraries of object-oriented functions designed to help the VR developer;
- Support various common i/o devices used in VR (so drivers need not be written by the developer);
- Allow import of CAD models (saves time), editing of shapes, specifying object hierarchies, collision detection and multi-level of detail, shading and texturing, run-time management;
- Have built-in networking functions for multi-user interactions, etc.

VR Toolkits can be classified by:

- ✓Whether text-based or graphical-programming;
- ✓The type of language used and the library size;
- ✓The type of i/o devices supported;
- ✓The type of rendering supported;
- ✓Whether general-purpose or application specific;
- ✓Whether proprietary (more functionality, better documented) or public domain (free, but less documentation and functionality)

VR Toolkits in Early 90s

- ***RenderWare*** (Cannon), ***VRT3/Superscape*** (Dimension Ltd.), ***Cyberspace Developer Kit*** (Autodesk), ***Cosmo*** Authoring Tool (SGI/Platinum/CA), ***Rend386*** and others;
- They allowed either text-based programming (RenderWare, CDK and Rend386), or graphical programming (Superscape and Cosmo);
- They were platform-independent and generally did not require graphics acceleration hardware;
- As a result they tended to use "low-end" I/O devices (mouse) and to support flat shading to maintain fast rendering.

Rend386 scene

VR Toolkits discussed in this chapter

Name	Application Area	Proprietary	Library size language
WorldToolKit (WTK) Sense8/EAI/EDS/)	General purpose	yes	"C" >1,000 functions
Java3D (Sun Microsystems)	General Purpose	no	Implemented in C Programming in Java 19 packages, 275 classes
GHOST (SensAble Technologies)	Haptics for Phantom	yes	C++
PeopleShop (Boston Dynamics)	Military/civilian	yes	C/C++

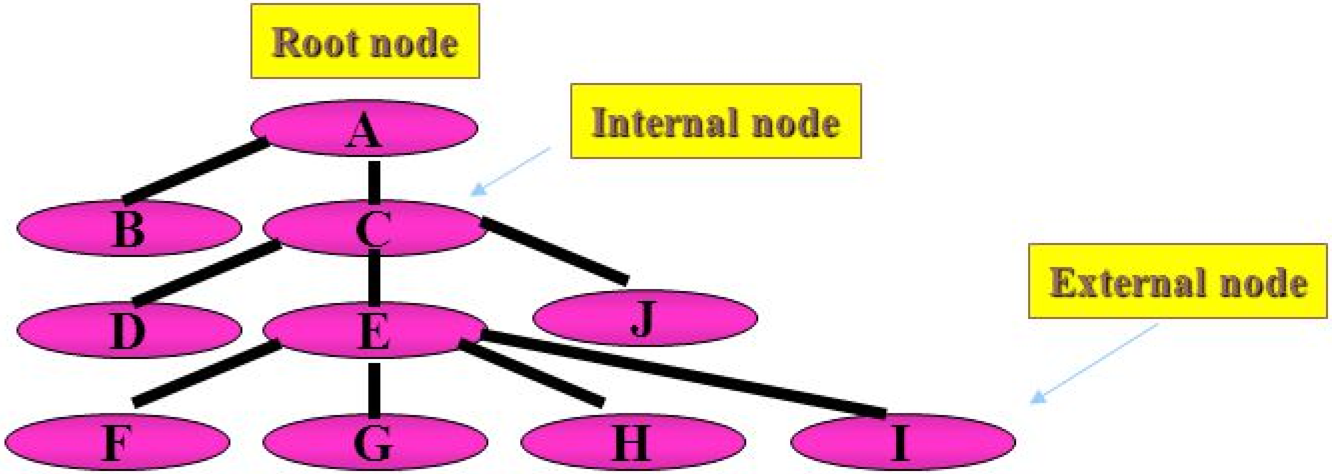
The scene graph:

✓Is a hierarchical organization of objects (visible or not) in the virtual world (or "universe") together with the view to that world;

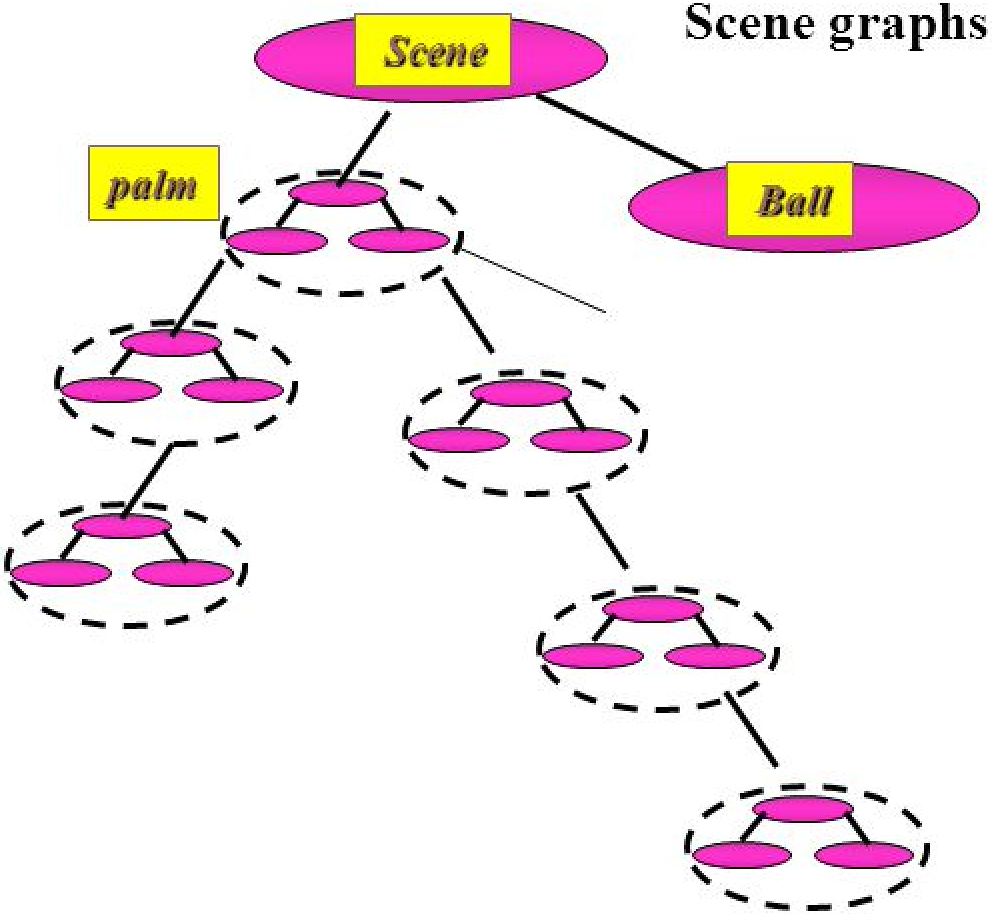
✓Scene graphs are represented by a tree structure, with nodes connected by branches.

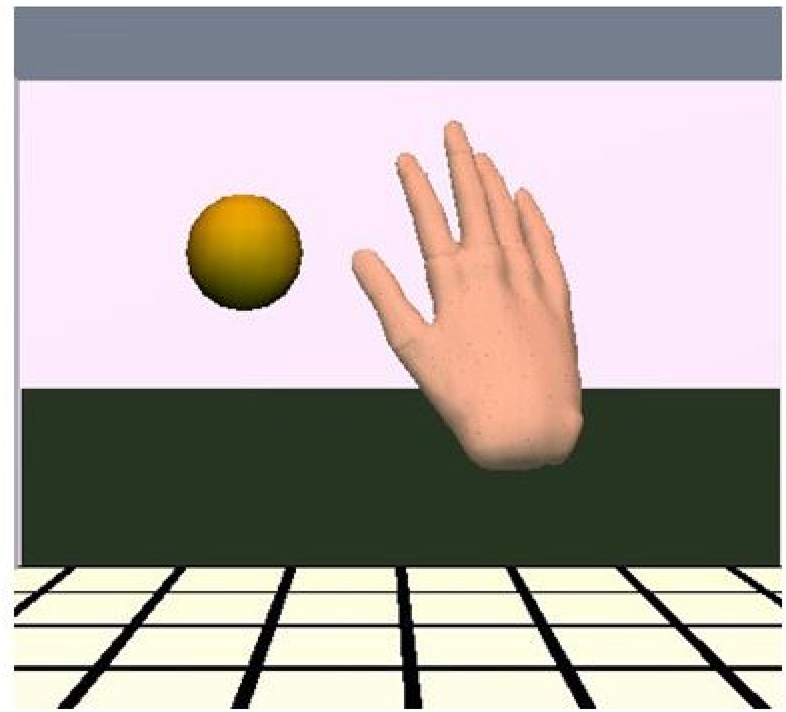
✓Visible objects are represented by external nodes, which are called leaves (they have no children). Example nodes F, G, H, I

✓Internal nodes represent transformations (which apply to all their children)

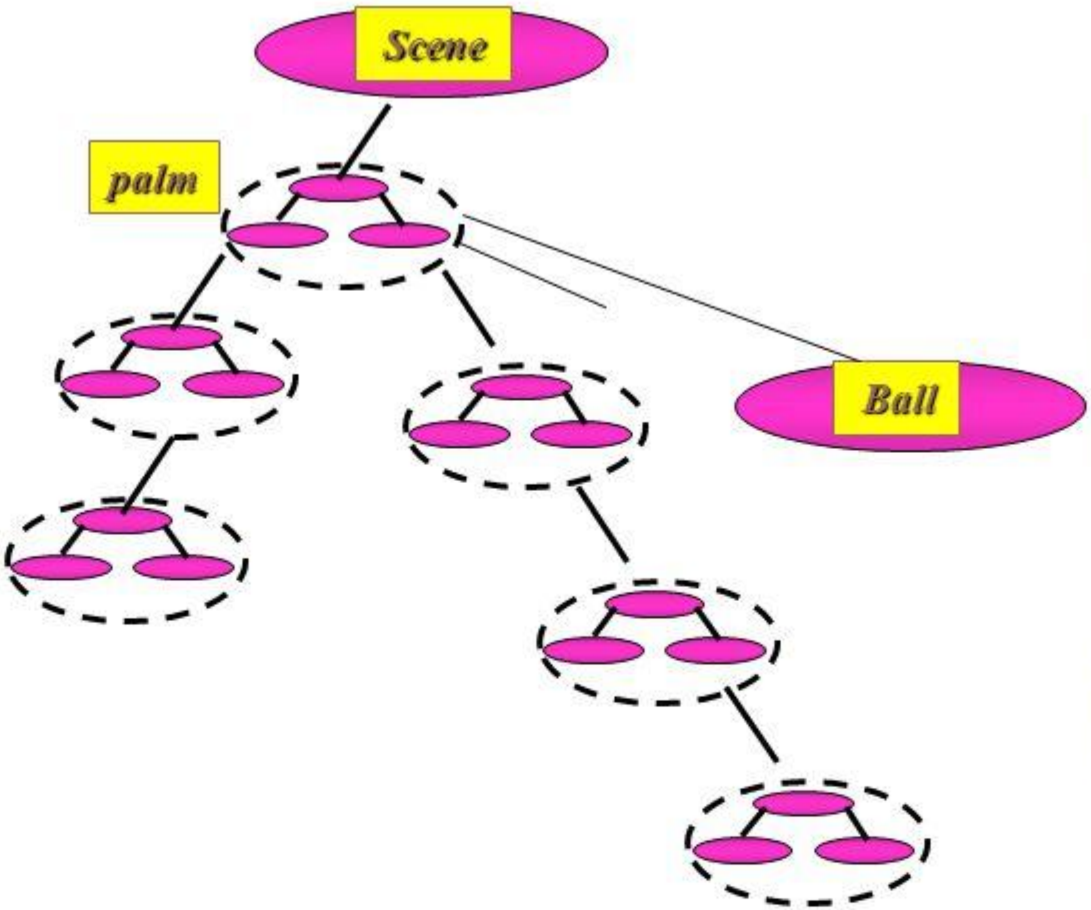


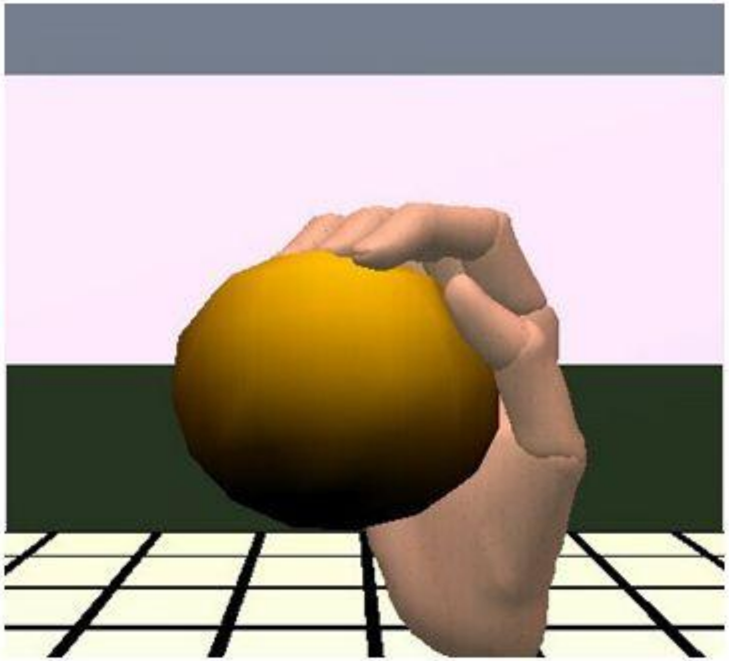
Scene graphs are not static





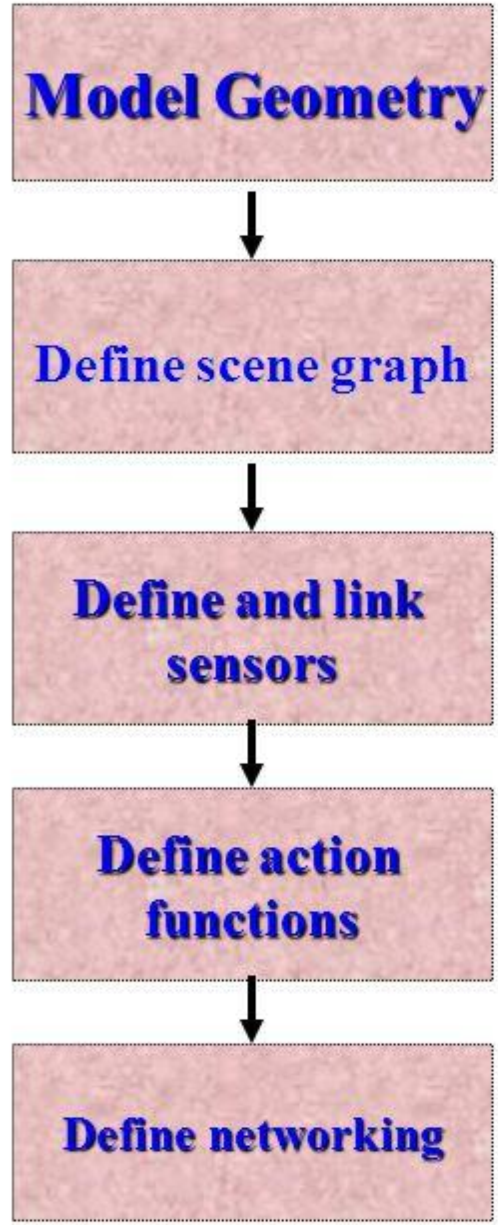
Scene graph shows that
the ball is a child of "scene"



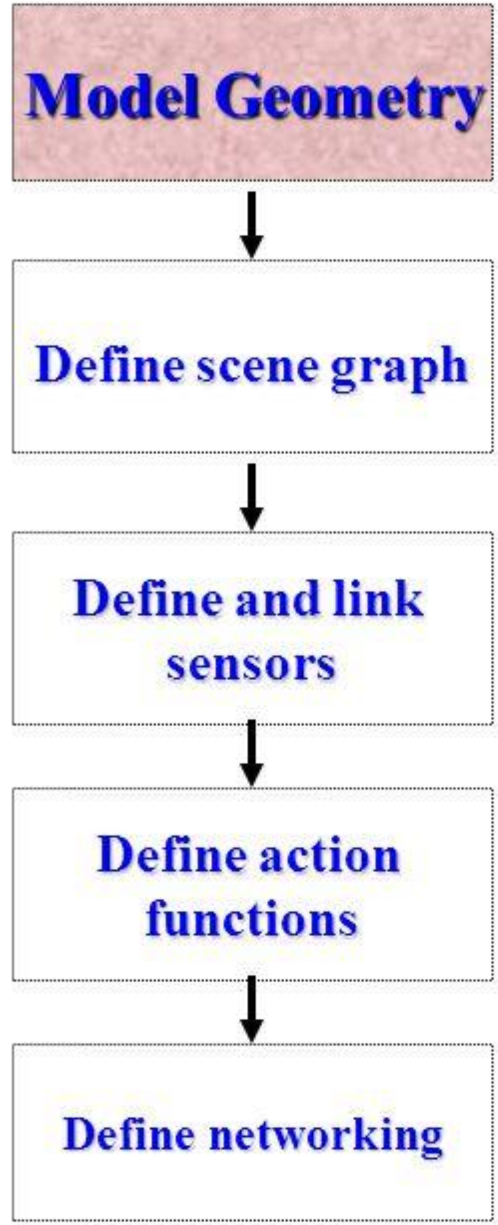


Scene graph has been modified, such that the ball is now a child of the palm

WTK Initiation



WTK Initiation



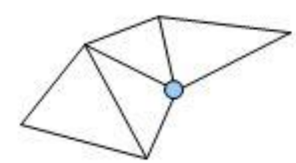
WTK geometry:

- Are the only visible objects in the scene (others like viewpoint, serial ports, etc; are not);
- Geometries are either imported from CAD (ex. dxf or 3ds formats), or from VRML (wrl) or through neutral file format (nff);
- Custom geometry created through polygons and vertices;

Imported geometry:
WTgeometrynode_load(hand)

Geometry primitive:
WTgeometry_newsphere()

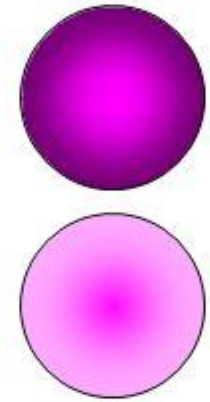
Custom geometry:
WTgeometry_begin
……
WTpoly_addvertex()
WTgeometry_save



WTK object appearance:

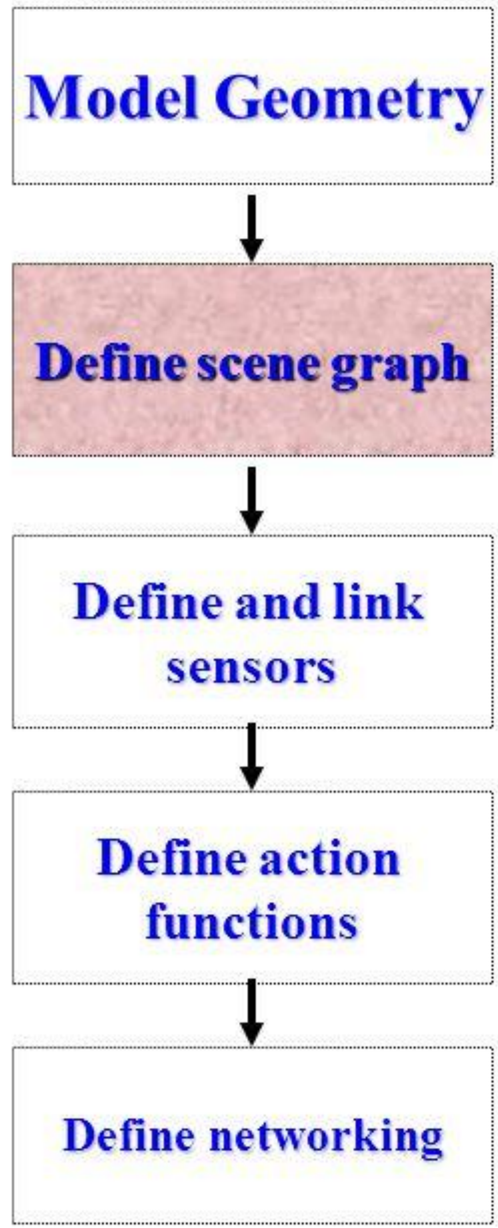
- Objects have material properties such as the way they reflect light (ambient, diffuse, specular, shininess, emissive, opacity); These properties are specified using material tables
- Textures are loaded from files or created and then filtered (scaled) to the object size

To load a material table:
WTmtable_load(filename)



Applying texture:
WTtexture_load
WTgeometry_settexture_
WTtexture_setfilter

WTK Initiation

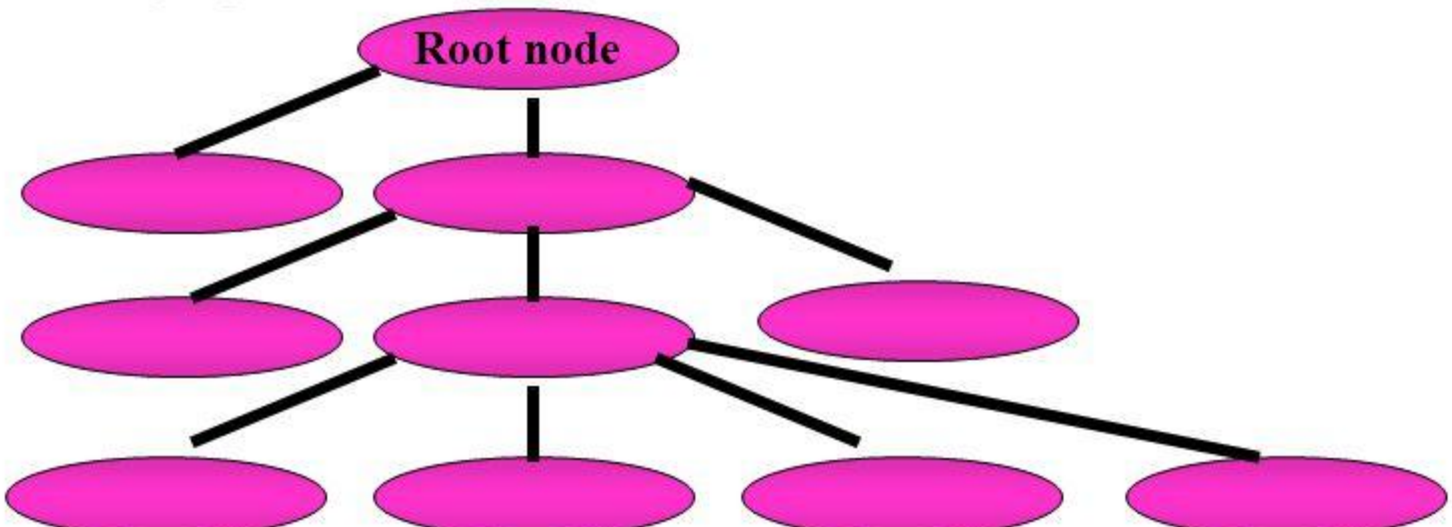


WTK scene graph:

✓The scene consists of various objects, some visible (geometry), some not (viewpoint, transforms, etc.); These objects are ***nodes*** in a scene graph;

✓The scene graph is the hierarchical arrangement of nodes that expresses the nodes spatial organization and relationship to each other.

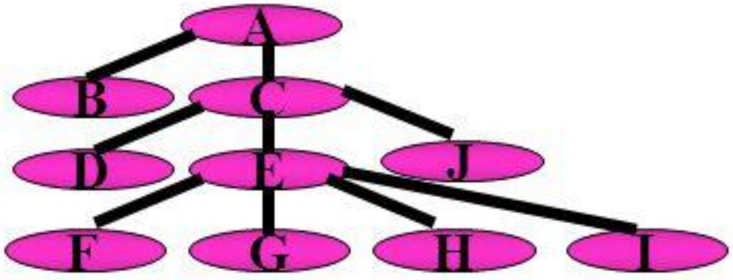
✓Each scene graph has only one root node.



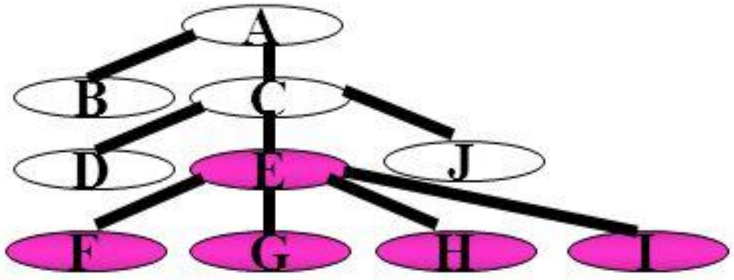
WTK scene graph terminology:

- If a node has a sub-tree that includes another node, it is its ***ancestor***. Example node A is ancestor of E;
- A ***parent*** node is a node direct ancestor. C is parent of E but not of I. C is an ancestor of I;
- ***Siblings*** are children nodes of the same parent. F,G,H,I are sibling nodes;
- If a node is rendered before another, it is its ***predecessor*** (need not be its ancestor). B is a predecessor of J, but not its ancestor. Node B can affect the rendering of node J.

Scene graph tree



Scene graph sub-tree

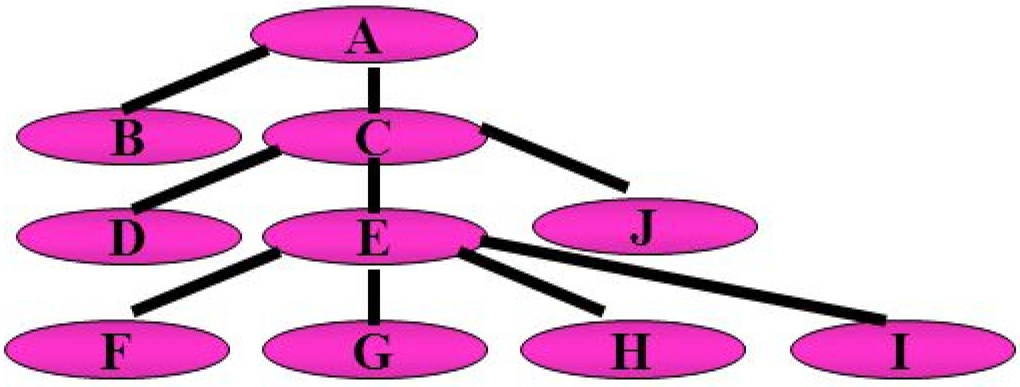


WTK scene graph traversal:

✓The order in which nodes appear in the scene graph determines the order in which they are rendered. This is because at each frame the scene graph is traversed top-to-bottom, left-to-right;

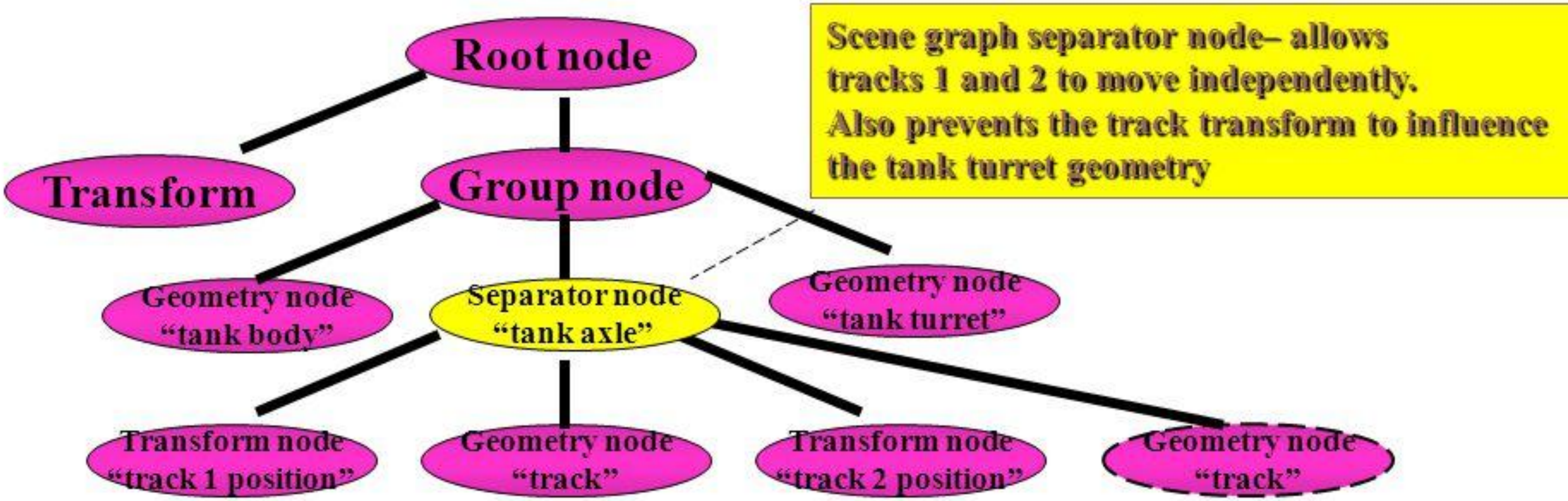
✓Advantages of using scene graph include object grouping, level-of-detail switching, instancing of geometry and sub-trees (better memory usage), increased frame rate (better culling), multiple scene graphs.

Traversal order is A, B, C…..



WTK node types:

- ***Geometry nodes*** – used for visible objects;
- ***Attribute nodes*** (fog, light, transform) – affect the way the geometry nodes are rendered; Need to be placed in the graph *before* the geometry they affect;
- ***Procedural nodes*** (root, level-of-detail, separator, switch, etc) – control the way the scene graph is processed

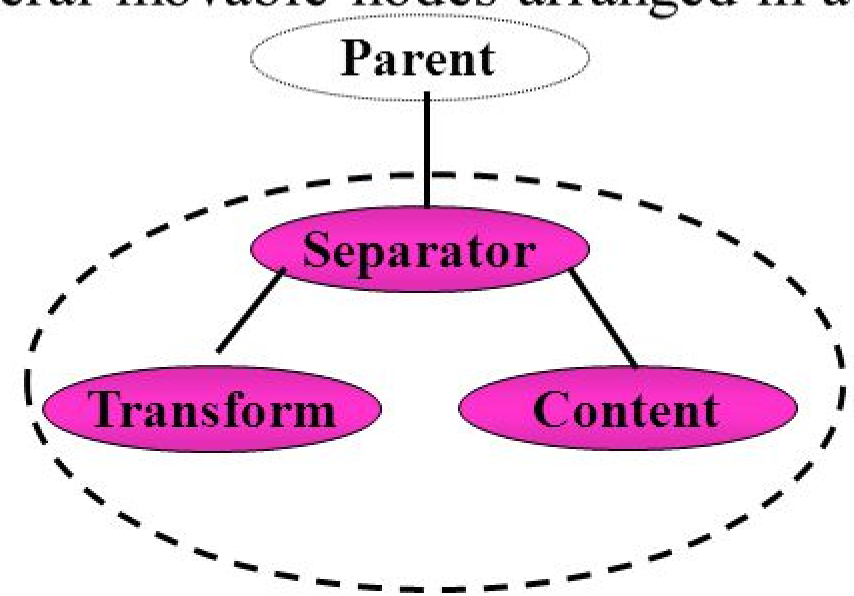


WTK movable node:

✓To help manage the state of the geometry nodes, and simplify scene graph construction, WTK has a self-contained kind of node called ***movable node***;

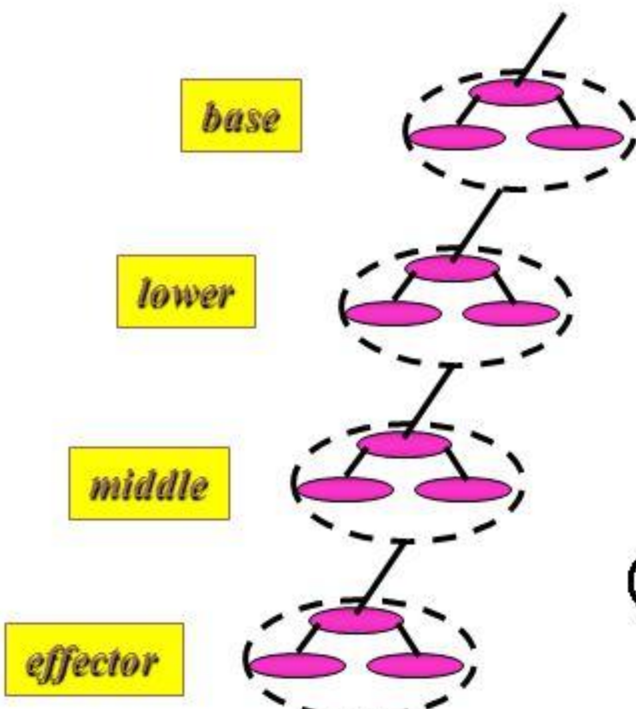
✓A movable node has its own separator, transform and content (geometry, light, switch, level-of-detail) nodes;

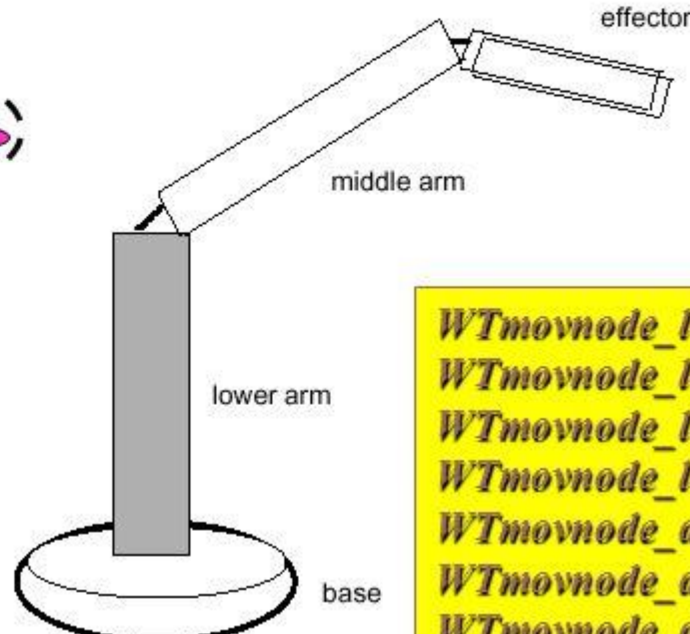
✓There can be several movable nodes arranged in a hierarchy



WTK movable node hierarchy:

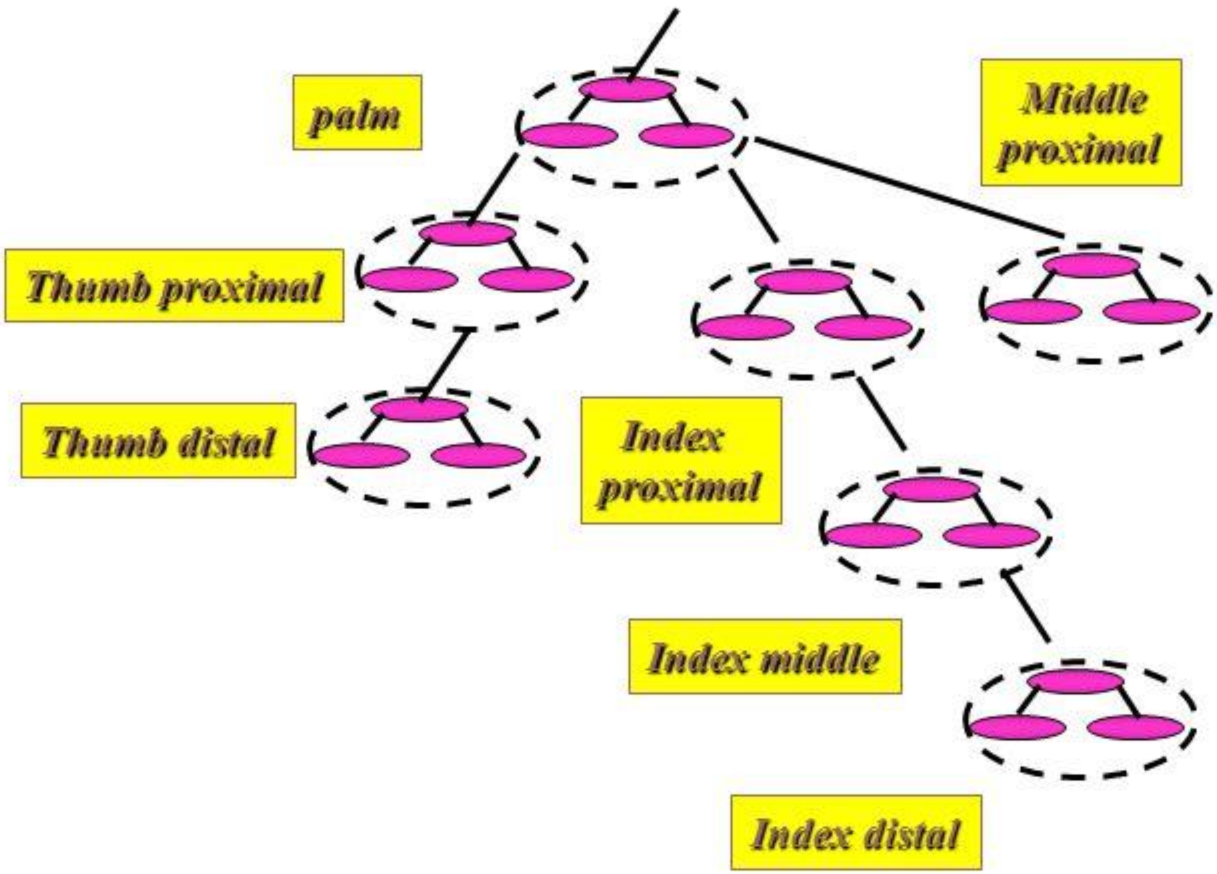
- To create a robot arm, each of the objects need to be created separately and loaded as movable nodes (base, lower arm, middle arm, effector);
- Then they need to be linked in a scene graph






```
WTmovnode_load(base)
WTmovnode_load(lower)
WTmovnode_load(middle)
WTmovnode_load(effector)
WTmovnode_attach(base,lower);
WTmovnode_attach(lower, middle);
WTmovnode_attach(middle, effector);
```

WTK virtual hand hierarchy:



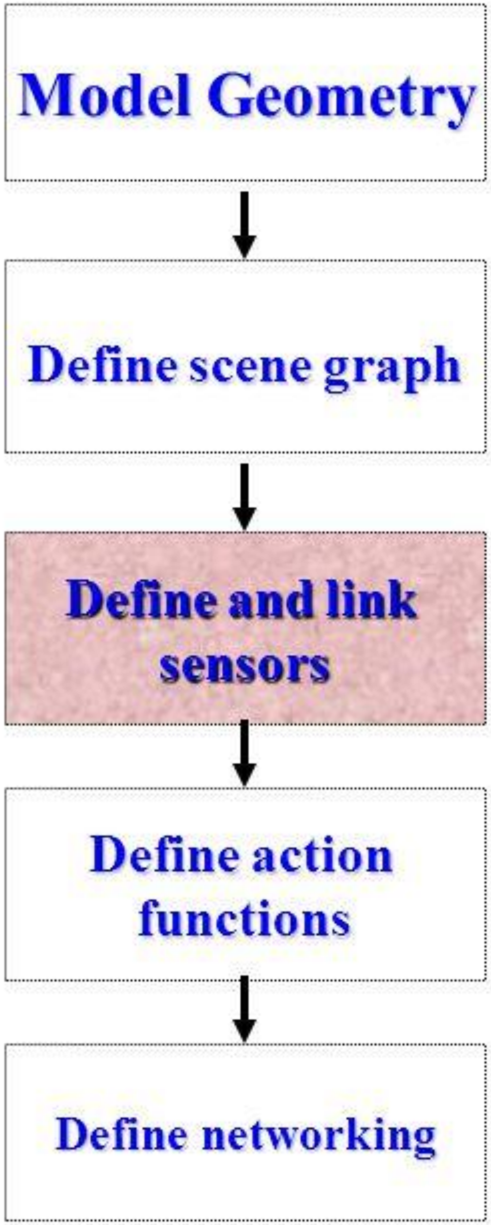
WTK virtual hand loading:

```
/* Load the hand model */
 Palm = WTmovnode_load(Root, "Palm.nff", 1.0);

 ThumbProximal = WTmovnode_load(Palm, "ThumbProximal.nff", 1.0);
 ThumbDistal = WTmovnode_load(ThumbProximal, "ThumbDistal.nff", 1.0);

 IndexProximal = WTmovnode_load(Palm, "IndexProximal.nff", 1.0);
 IndexMiddle = WTmovnode_load(IndexProximal, "IndexMiddle.nff", 1.0);
 IndexDistal = WTmovnode_load(IndexMiddle, "IndexDistal.nff", 1.0);

 MiddleProximal = WTmovnode_load(Palm, "MiddleProximal.nff", 1.0);
 MiddleMiddle = WTmovnode_load(MiddleProximal, "MiddleMiddle.nff", 1.0);
 MiddleDistal = WTmovnode_load(MiddleMiddle, "MiddleDistal.nff", 1.0);

 RingProximal = WTmovnode_load(Palm, "RingProximal.nff", 1.0);
 RingMiddle = WTmovnode_load(RingProximal, "RingMiddle.nff", 1.0);
 RingDistal = WTmovnode_load(RingMiddle, "RingDistal.nff", 1.0);

 SmallProximal = WTmovnode_load(Palm, "SmallProximal.nff", 1.0);
 SmallMiddle = WTmovnode_load(SmallProximal, "SmallMiddle.nff", 1.0);
 SmallDistal = WTmovnode_load(SmallMiddle, "SmallDistal.nff", 1.0);
```

WTK virtual hand hierarchy:

```
WTmovnode_attach(Palm,ThumbProximal);
WTmovnode_attach(ThumbProximal, ThumbDistal);
WTmovnode_attach(Palm, IndexProximal);
WTmovnode_attach(IndexProximal, IndexMiddle);
WTmovnode_attach(IndexMiddle, IndexDistal);
WTmovnode_attach(Palm, MiddleProximal);
WTmovnode_attach( MiddleProximal, MiddleMiddle);
WTmovnode_attach(MiddleMiddle, MiddleDistal);
WTmovnode_attach(Palm, RingProximal);
WTmovnode_attach( RingProximal, RingMiddle);
WTmovnode_attach(RingMiddle, RingDistal);
WTmovnode_attach(Palm, SmallProximal);
WTmovnode_attach( SmallProximal, SmallMiddle);
WTmovnode_attach(SmallMiddle, SmallDistal);
```

WTK Initiation

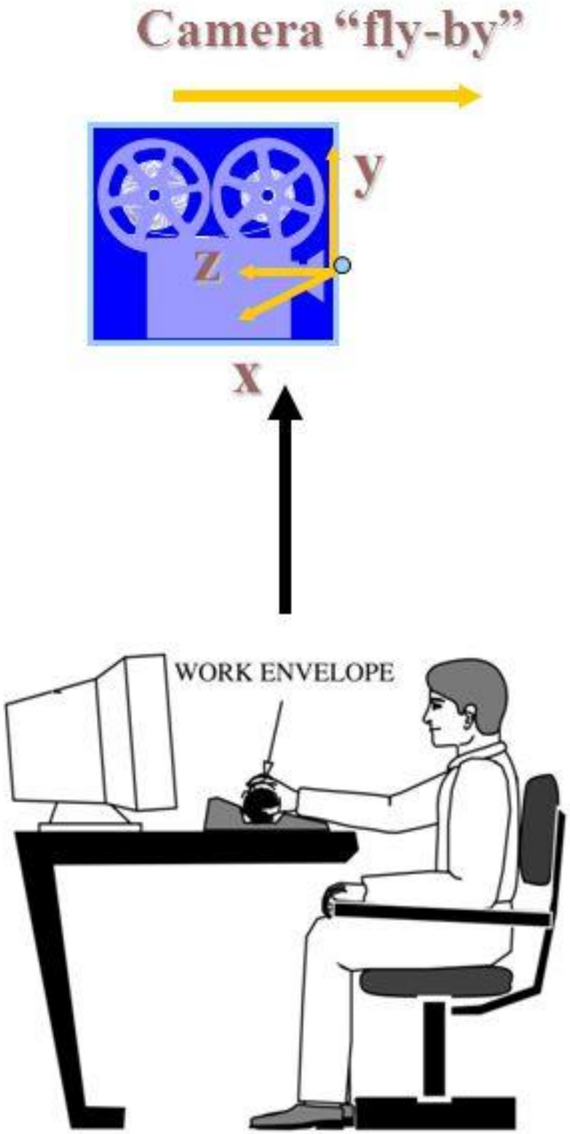


WTK sensors:

✓Allow the user to interact dynamically with the simulation by providing input and receiving feedback from the simulation. Some of the supported sensors are:

- track balls (spaceball, geometry ball Jr);
- trackers (Polhemus Fastrack, Isotrack, Insidetrack; Ascension Bird and Flock of Birds);
- sensing gloves (5DT serial glove, Pinch glove, CyberGlove);
- displays (CrystalEyes glasses, BOOM display, Virtual i/o HMD, CyberMaxx2 HMD)

Etc.



Using the trackball:

- We can use the spaceball to interactively change the viewpoint to the scene;
- The spaceball needs to be declared as a sensor and needs to linked to the serial port;
- Then the sensor needs to be attached to the viewpoint.

```
main() {
 WTsensor  *spaceball;
  Wtnode *root, *scene;

/* initialize the universe*/
WTuniverse_new(WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT);

/*load scene at the root*/
root= WTuniverse_getrootnodes();
Scene=Wtnode_load(root,"myscene", 1.0);

/*attach sensor to the serial port*/
spaceball=WTspaceball_new(SERIAL2);

/*attach viewpoint to the spaceball*/
WTviewpoint_addressor(Wtuniverse_getviewpoints(), spaceball);

/* start simulation */
WTuniverse_ready();
WTuniverse_go();
/*stop simulation*/
WTuniverse_delete();
return0;
}
```

WTK Initiation

Model Geometry

↓

Define scene graph

↓

Define and link sensors

↓

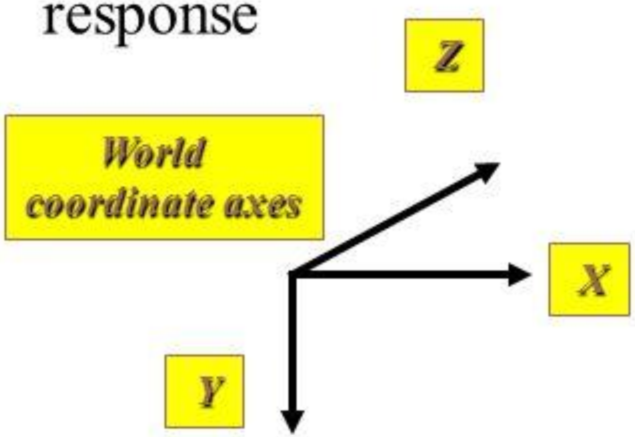
Define action functions

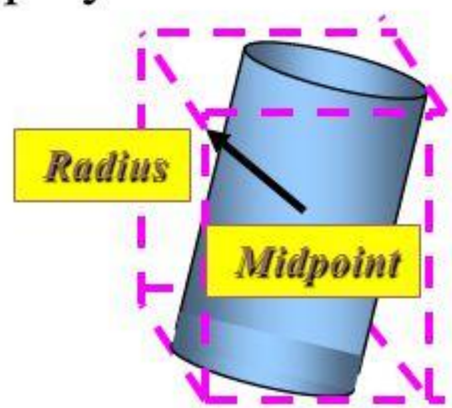
↓

Define networking

WTK action functions:

- To do the ball grasping we need to check for collision between the hand and the ball, and then we need to make the ball a child of the palm.
- WTK action functions are user defined functions that are executed at every simulation loop (frame). Such functions are collision detection and collision response.
- In our case also sound needs to be played as a form of collision response



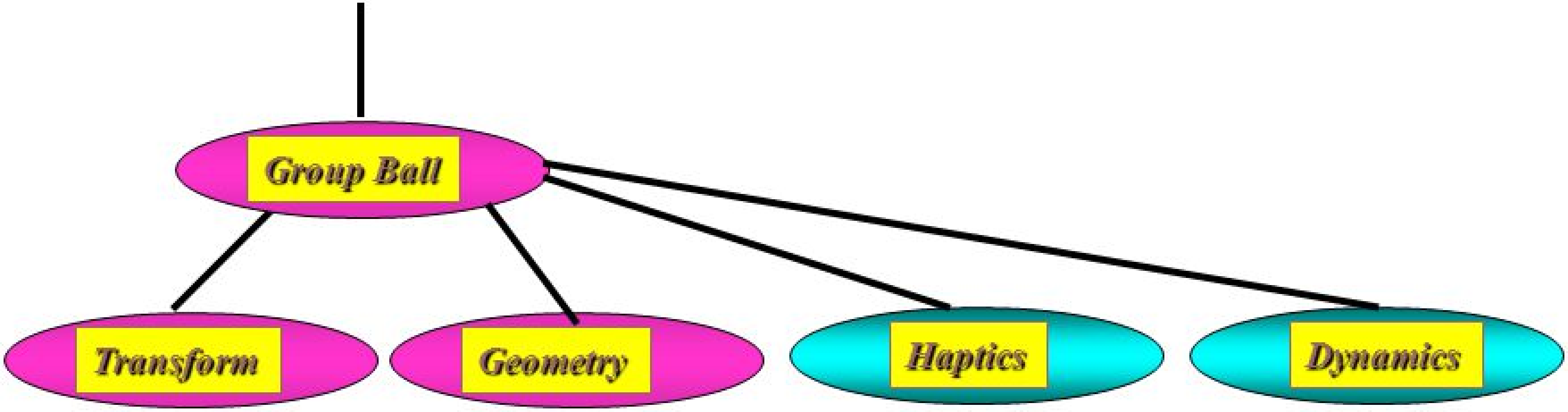


WTK action functions:

```
WTsound_load("spring");

void action()
{
 /* Check for collision detection */
 if(WTnodepath_intersectbbox(HandNP, BallNP))
 {
  /* play spring sound*/
  WTsound_play(spring);

  /* Remove the Ball from the scene graph and immediately reattach it
     as a child of the Palm
  */
  WTnode_remove(Ball);
  WTmovnode_attach(Palm, Ball, 0);

  /* stop playing spring sound */
  WTsound_stop(spring)
 }
}
```

WTK scene graph extension – the haptic node:

- Another form of collision response is force feedback if the user has a haptic glove (such as Rutgers Master II);
- This is compatible with VRML;
- The fields of the haptics node are stiffness, viscosity, friction and haptic effect (indicating a force profile – square, sine, constant, ramp)



from (Popescu, 2001)

WTK Initiation

Model Geometry

↓

Define scene graph

↓

Define and link sensors

↓

Define action functions

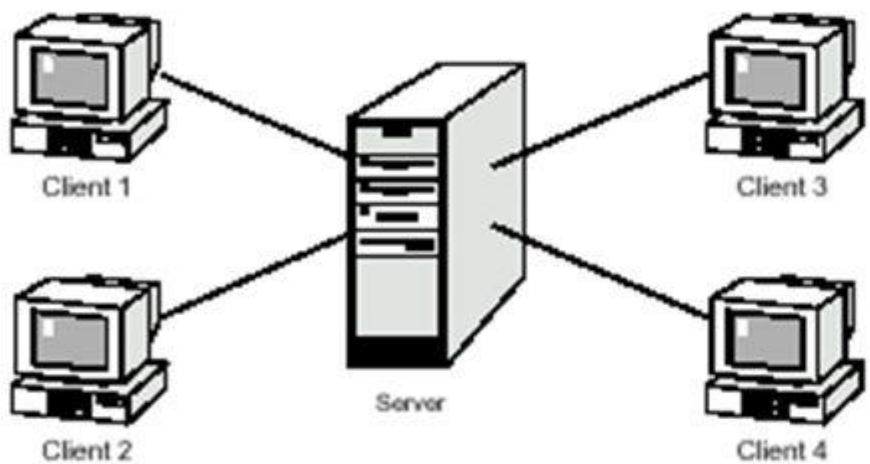
↓

Define networking

WTK networking:

✓Uses the "World2World" library extension of WTK;

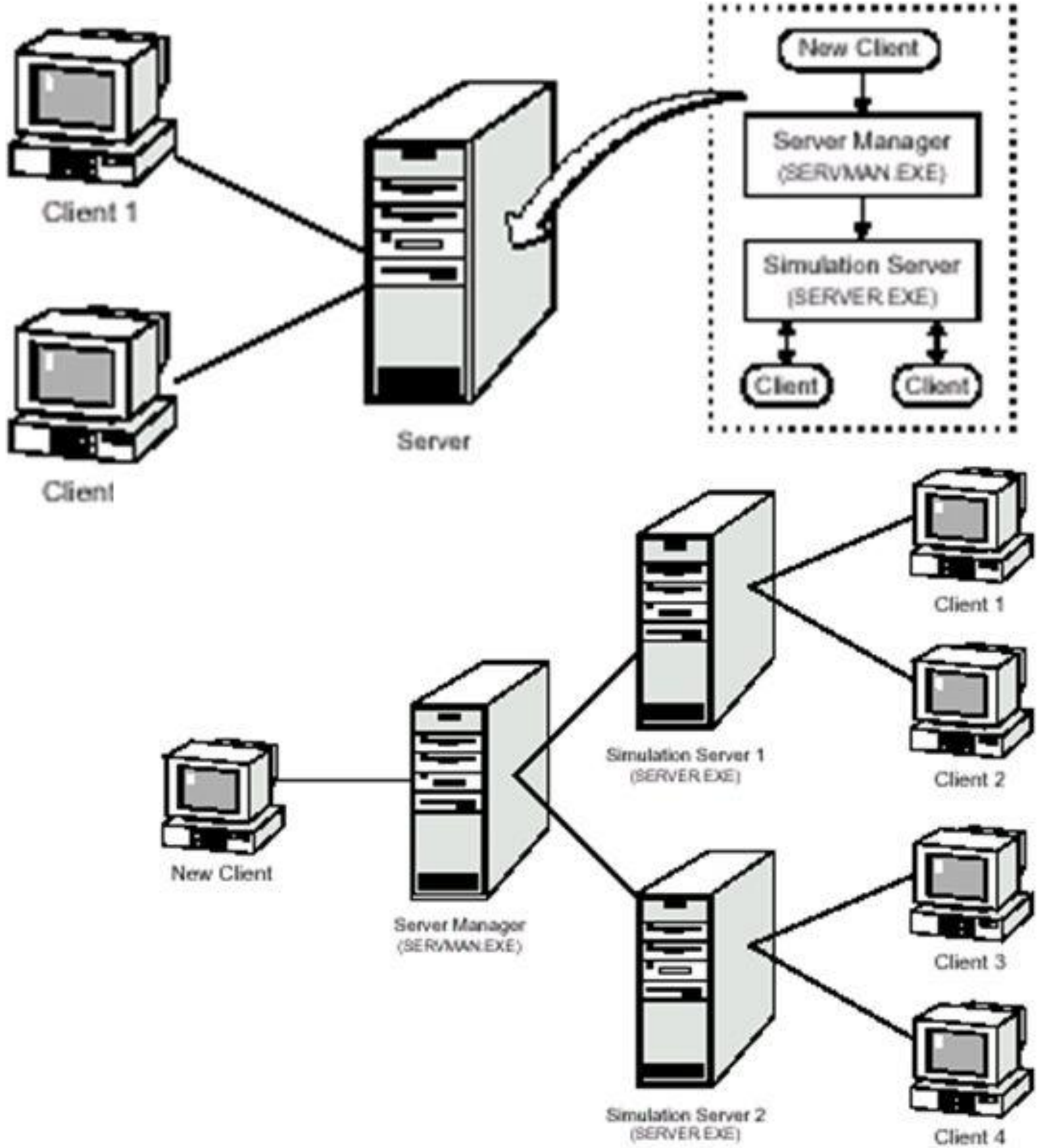
✓A typical client-server architecture uses a single server that does "double duty" managing connections as well as data sharing. Simulation stops when a new client requests connection.



Typical client-server architecture

✓WTK/W2W uses a single server ***manager*** and several ***simulation servers*** to improve scalability and system response:

- The server manager is the initial point of contact of a new client connecting to the simulation – administration tasks performed transparently of the simulation;
- The simulation servers interact directly with the assigned clients, once handed over by the manager.
- This way the ongoing simulation is not disrupted when a new client is requesting connection.



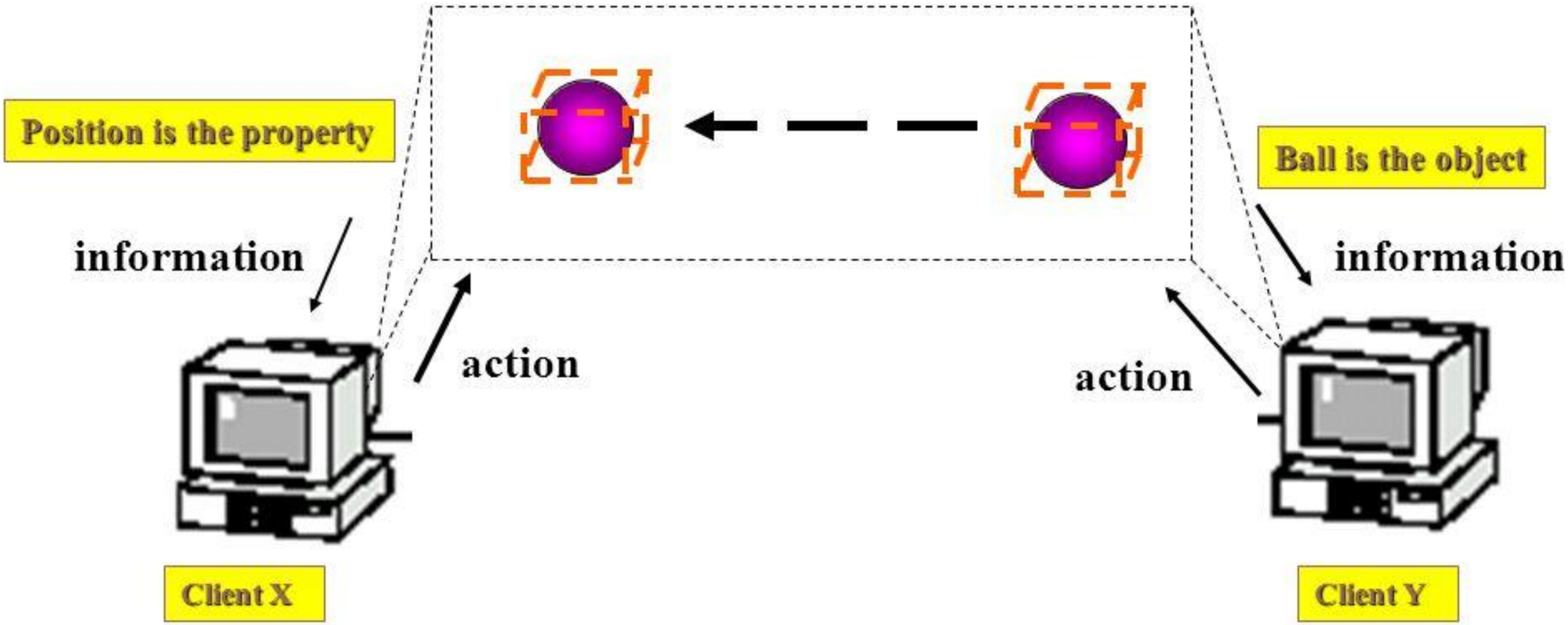
WTK two-tier client-server architecture

(from World2World release 1, Sense8 Co.)

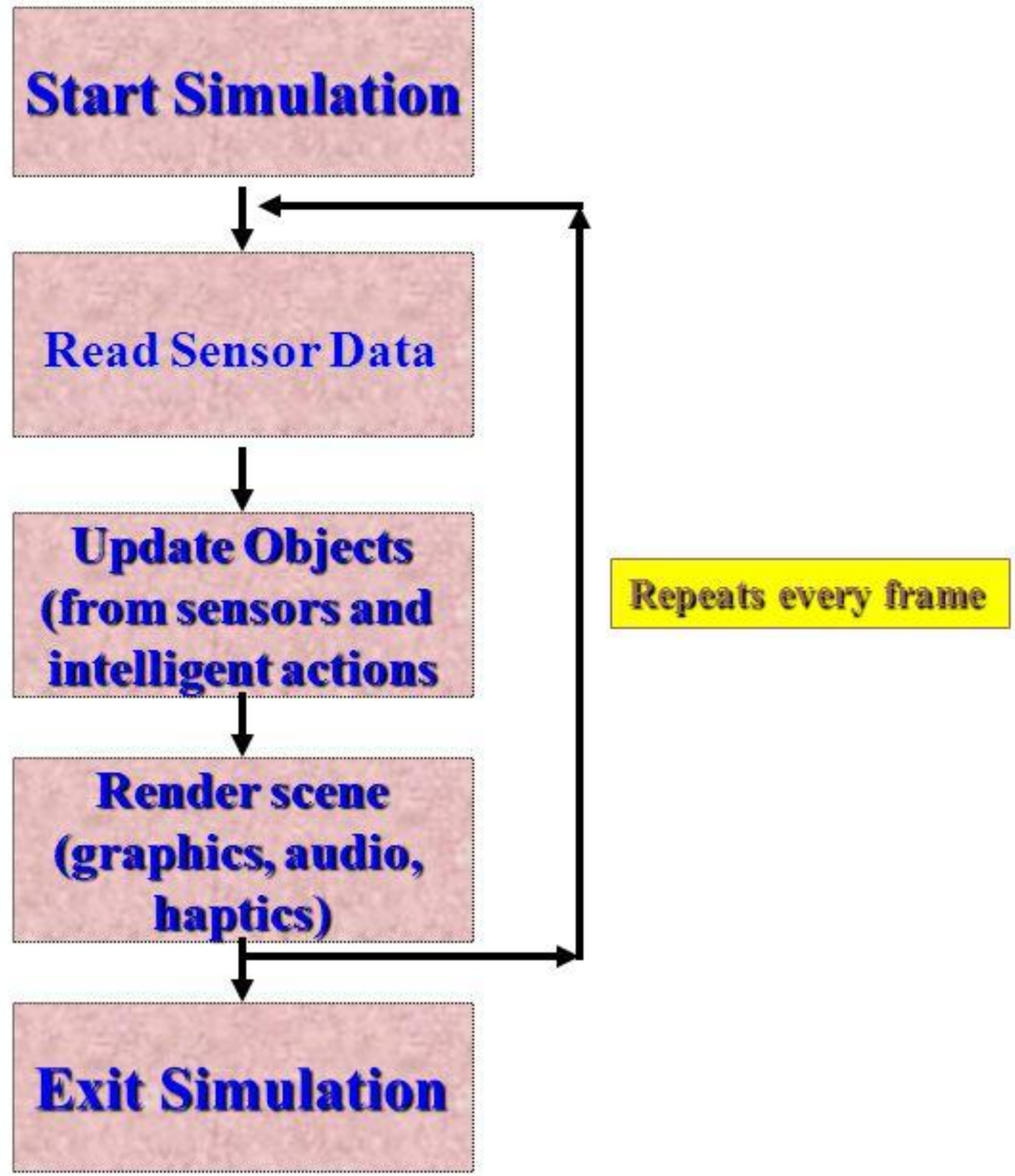
WTK Simulation Servers:

✓ Shared properties are organized in shared groups. Client X and Client Y are interested in the position property of the ball object.

✓ The simulation Server manages the distribution of shared properties to clients that registered interest in that shared group



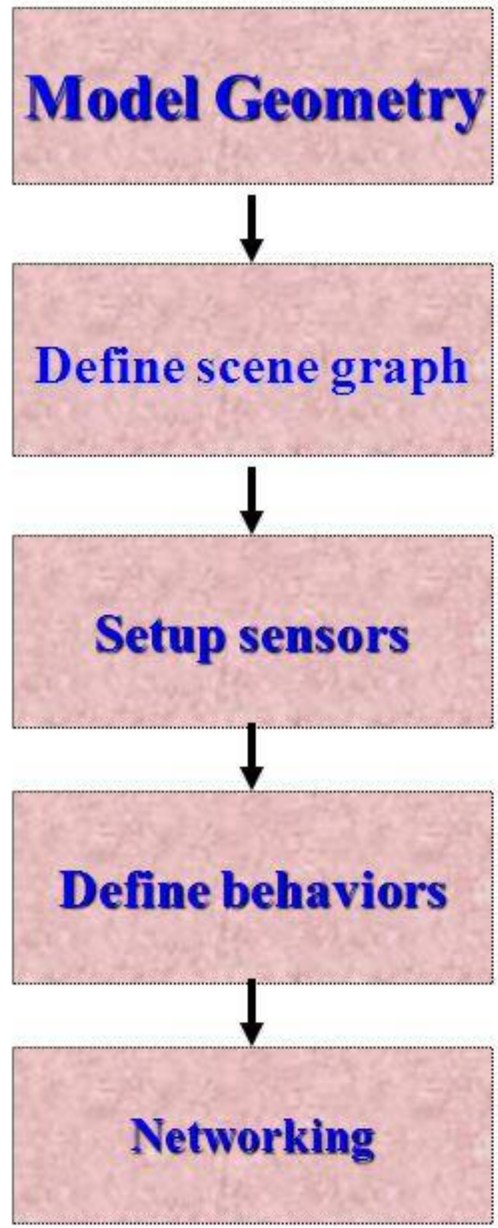
WTK Run-time loop



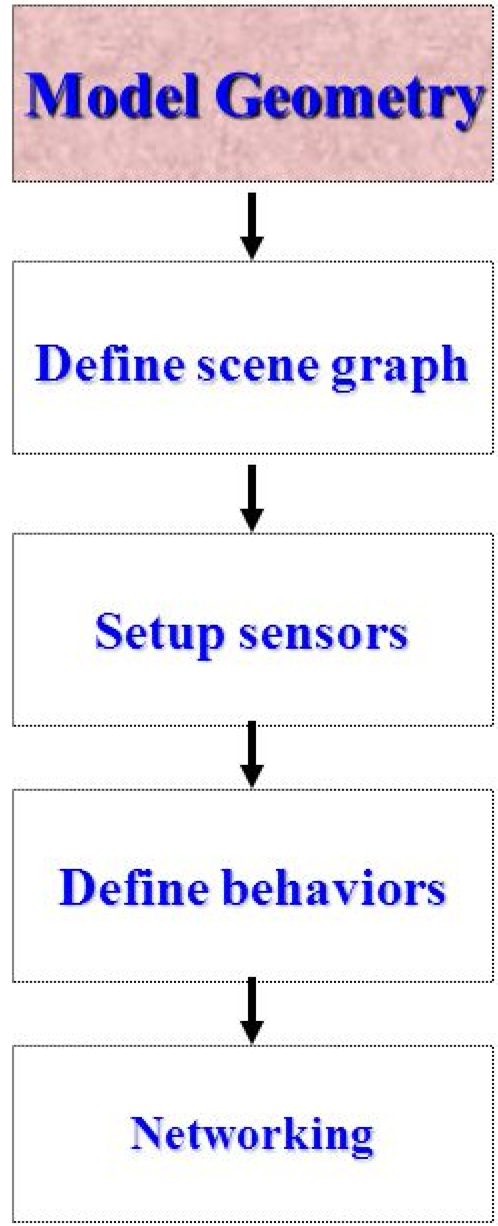
Java and Java 3D

- Java
  - object oriented programming language
  - developed for network applications
  - platform independence
  - slower than C/C++
- ***Java 3D***
  - Java hierarchy of classes that serves as an interface to 3D graphics rendering and sound rendering systems
  - Perfectly integrated with Java
  - Strong object oriented architecture
  - Powerful 3D graphics API

Java 3D Initiation



Java 3D Initiation



Java 3D geometry:

- ✓Geometry can be imported from various file formats (e.g. 3DS, DXF, LWS, NFF, OBJ, VRT, VTK, WRL)
- ✓Can be created as a primitive geometry (e.g. sphere, cone, cylinder, …)
- ✓Custom geometry created by specifying the vertices, edges, normals, texture coordinates using specially defined classes

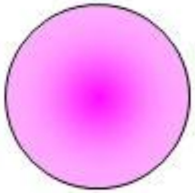
Imported geometry
loader.load("Hand.wrl")

Geometry primitive:
new Sphere(radius)

Custom geometry:
new GeometryArray(…)
new LineArray(…)
new QuadArray(…)
new TriangleArray(…)

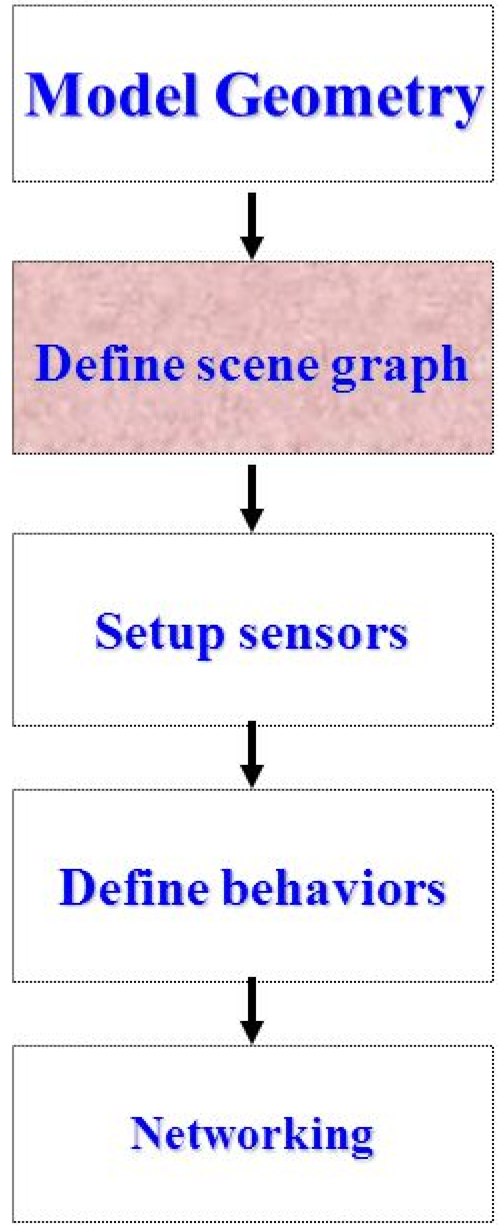
Java 3D object appearance:

- The appearance of a geometry is specified using an appearance object
- An appearance-class object stores information about the material (diffuse, specular, shininess, opacity, …) and texture

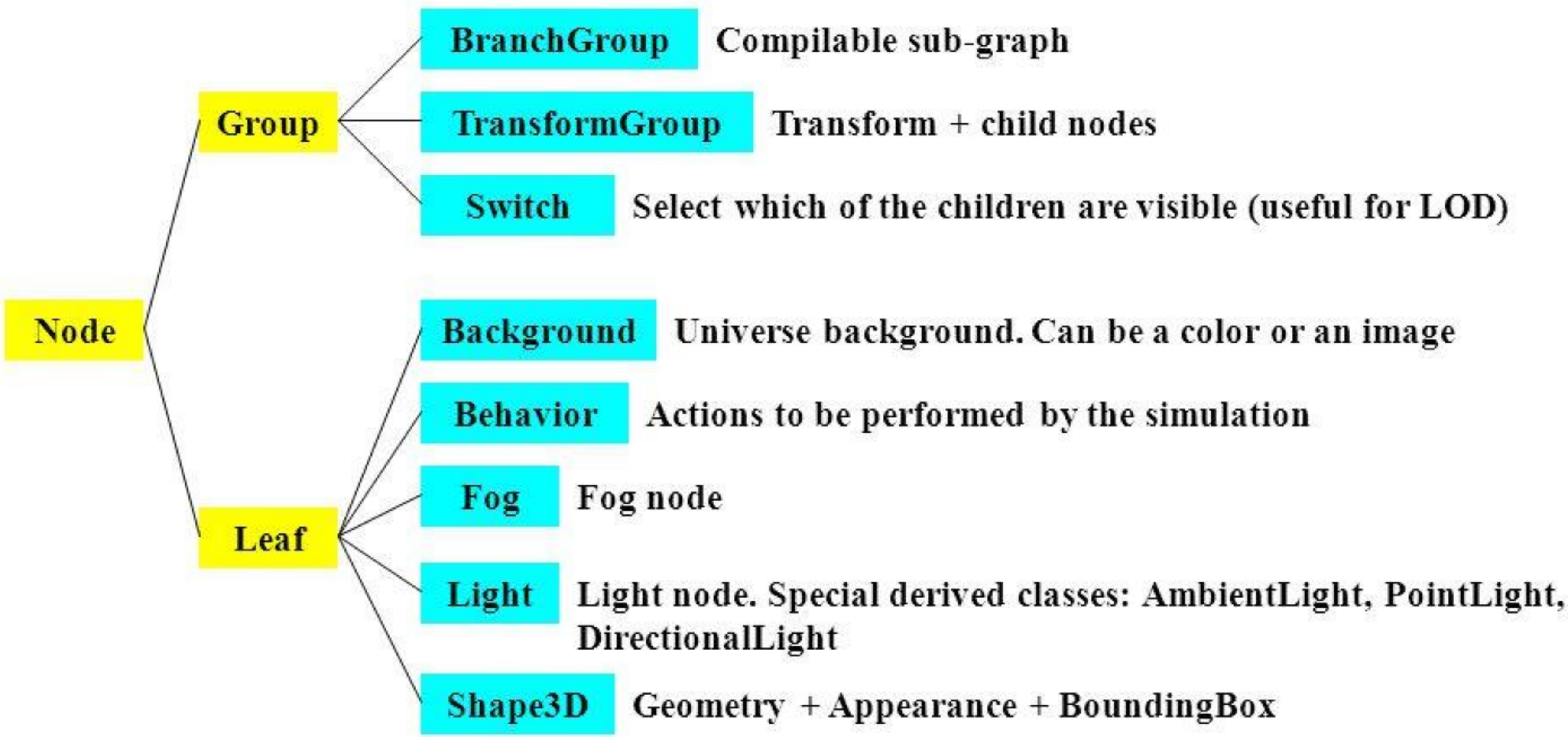


```
Mat = new Material();
Mat.setDiffuseColor(r, g, b);
Mat.setAmbientColor(r, g, b);
Mat.setSpecularColor(r, g, b);

TexLd = new TextureLoader("checkered.jpg", ...);
Tex = TexLd.getTexture();

Appr = new Appearance();
Appr.setMaterial(Mat);
Appr.setTexture(Text);

Geom.setAppearance(Appr)
```

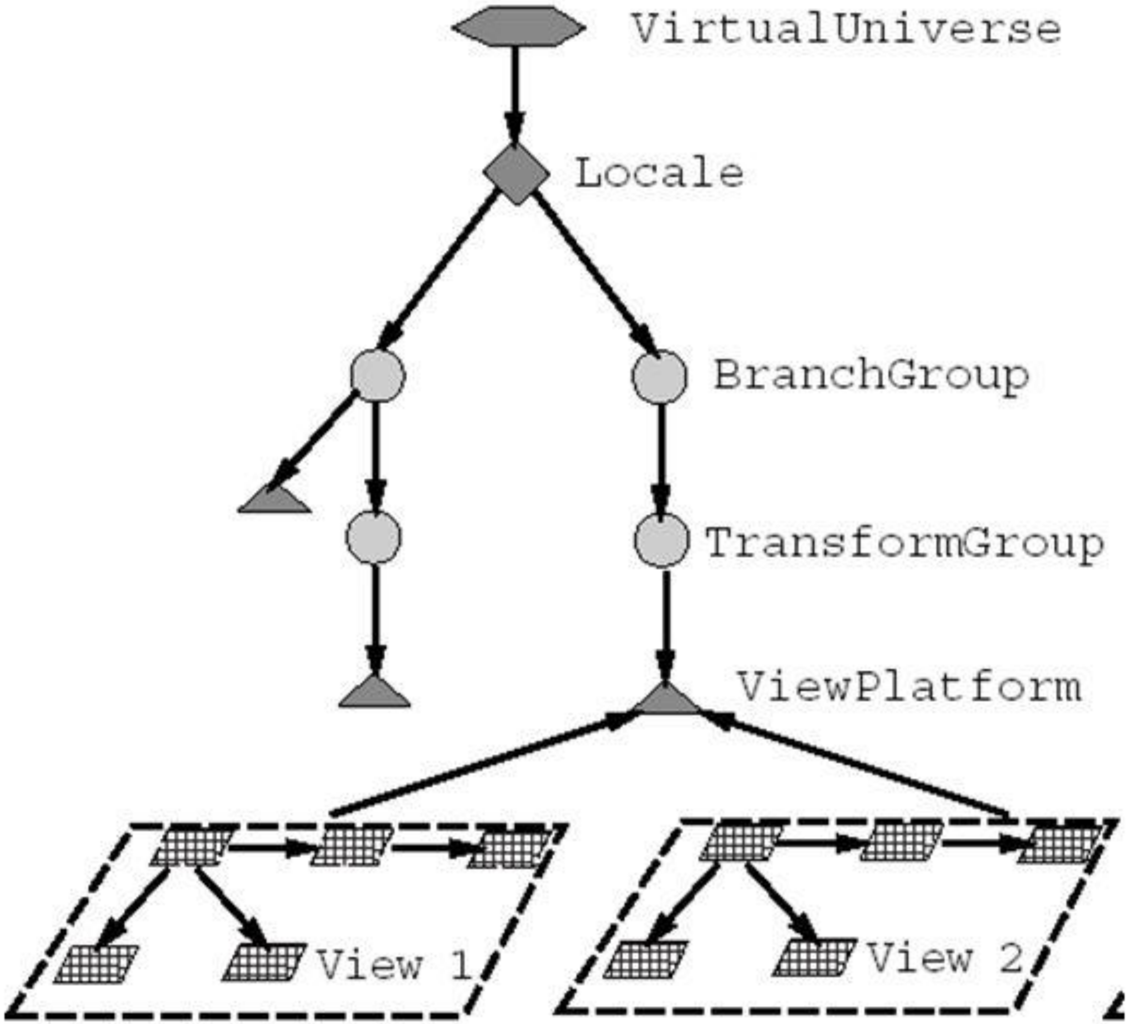
Java 3D Initiation



Java3D node types:



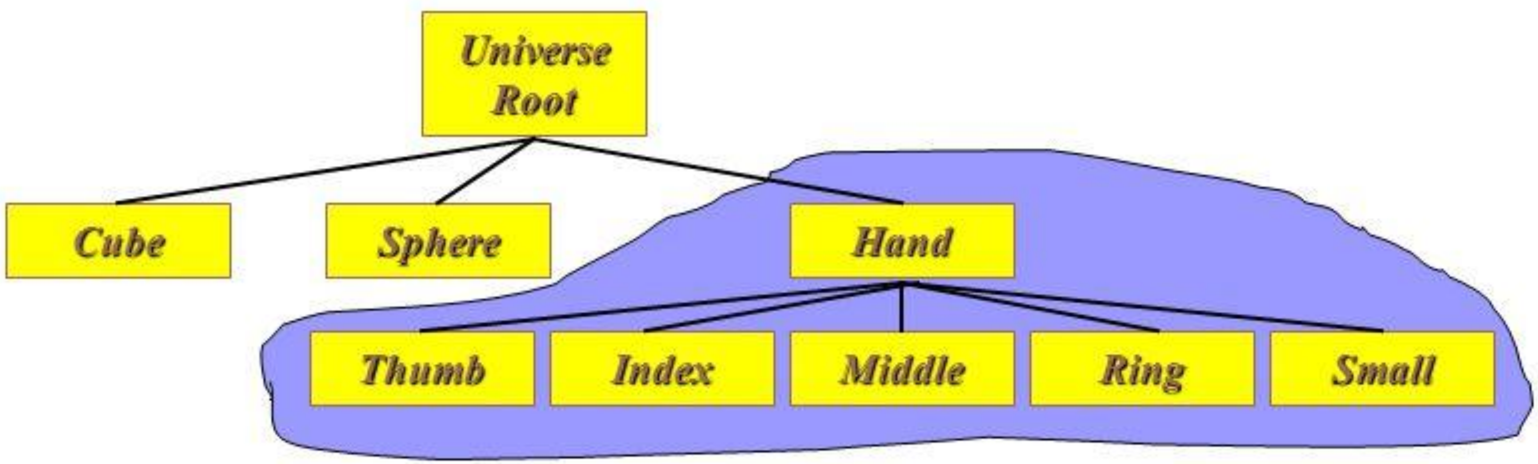
Java3D scene graph



Node

Loading objects from files

- Java3D offers by default support for Lightwave and Wavefront model files
- Loaders for other file formats can be downloaded for free from the web http://www.j3d.org/utilities/loaders.html
- Loaders add the content of the read file to the scene graph as a single object. However, they provide functions to access the subparts individually



Java3D model loading

Adding the model to the scene graph

```
Scene Sc = loader.load("Hand.wrl");
BranchGroup Bg = Sc.getSceneGroup();
RootNode.addChild(Bg);
```

Accessing subparts of the loaded model

```
Scene Sc = loader.load("Hand.wrl");
BranchGroup Bg = Sc.getSceneGroup();
Thumb = Bg.getChild(0);
Index = Bg.getChild(1);
Middle = Bg.getChild(2);
Ring = Bg.getChild(3);
Small = Bg.getChild(4);
```

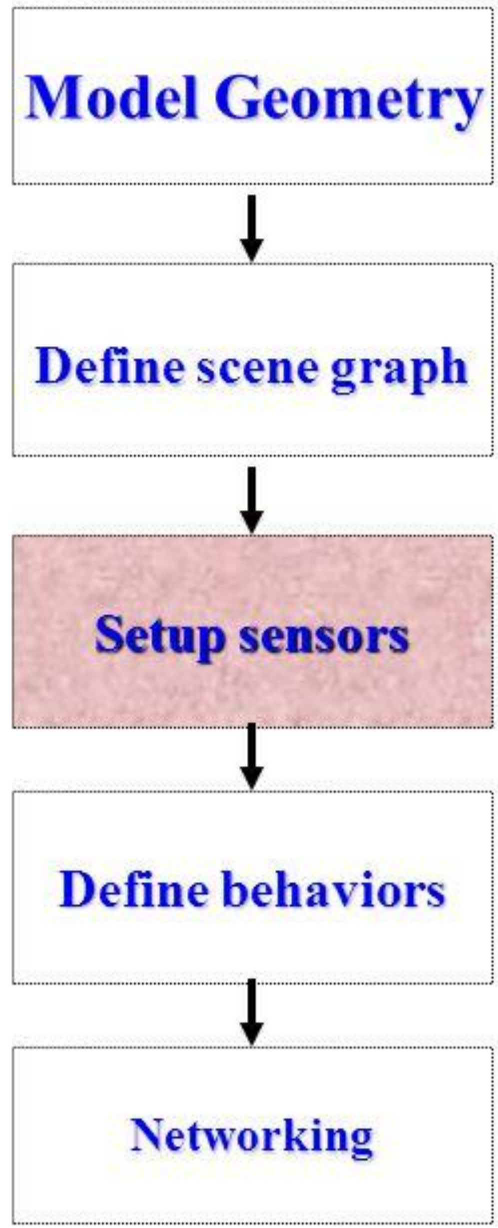
Java3D virtual hand loading:

```
Palm = loader.load("Palm.wrl").getSceneGroup();
ThumbProximal = loader.load("ThumbProximal.wrl").getSceneGroup();
ThumbDistal = loader.load("ThumbDistal.wrl").getSceneGroup();
IndexProximal = loader.load("IndexProximal.wrl").getSceneGroup();
IndexMiddle = loader.load("IndexMiddle.wrl").getSceneGroup();
IndexDistal = loader.load("IndexDistal.wrl").getSceneGroup();
MiddleProximal = loader.load("MiddleProximal.wrl").getSceneGroup();
MiddleMiddle = loader.load("MiddleMiddle.wrl").getSceneGroup();
MiddleDistal = loader.load("MiddleDistal.wrl").getSceneGroup();
RingProximal = loader.load("RingProximal.wrl").getSceneGroup();
RingMiddle = loader.load("RingMiddle.wrl").getSceneGroup();
RingDistal = loader.load("RingDistal.wrl").getSceneGroup();
SmallProximal = loader.load("SmallProximal.wrl").getSceneGroup();
SmallMiddle = loader.load("SmallMiddle.wrl").getSceneGroup();
SmallDistal = loader.load("SmallDistal.wrl").getSceneGroup();
```

Java3D virtual hand hierarchy:

```
Palm.addchild(ThumbProximal );
ThumbProximal .addchild(ThumbDistal );

Palm.addchild(IndexProximal );
IndexProximal .addchild(IndexMiddle );
IndexMiddle .addchild(IndexDistal );

Palm.addchild(MiddleProximal );
MiddleProximal .addchild(MiddleMiddle );
MiddleMiddle .addchild(MiddleDistal );

Palm.addchild(RingProximal );
RingProximal .addchild(RingMiddle );
RingMiddle .addchild(RingDistal );

Palm.addchild(SmallProximal );
SmallProximal .addchild(SmallMiddle );
SmallMiddle .addchild(SmallDistal );
```

Java3D Initiation



Input devices in Java3D

- The only input devices supported by Java3D are the mouse and the keyboard
- The integration of the input devices currently used in VR applications (position sensors, track balls, joysticks…) relies entirely on the developer
- Usually the drivers are written in C/C++. One needs either to re-write the driver using Java or use JNI (Java Native Interface) to call the C/C++ version of the driver. The latter solution is more desirable.
- Java3D provides a nice general purpose input device interface that can be used to integrate sensors. However, many times developers prefer custom made approaches

Java3D General purpose sensor interface

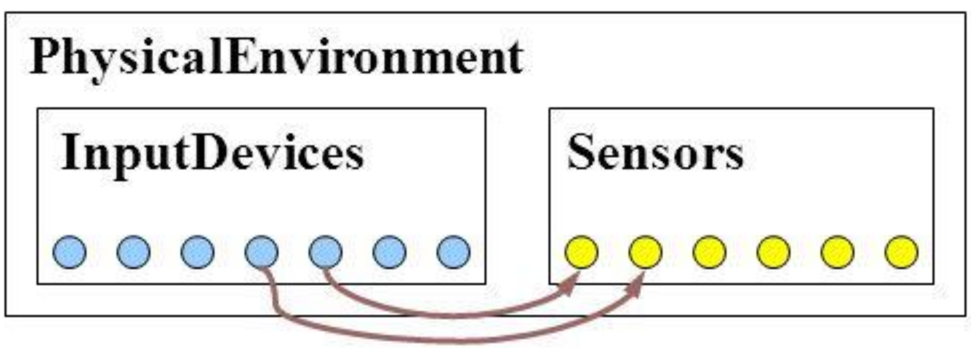
class PhysicalEnvironment - stores information about all the input devices and sensors involved in the simulation

class InputDevice - interface for an input device driver

class Sensor - class for objects that provide real time data

One input device can provide one or more sensors

A sensors object needs not be in relation with an input device (VRML style sensors)



Java3D Initiation

- Model Geometry
- ↓
- Define scene graph
- ↓
- Setup sensors
- ↓
- **Animating the scene**
- ↓
- Networking

Java3D - Animating the simulation

- Java3D offers Behavior objects for controlling the simulation
- A Behavior object contains a set of actions performed when the object receives a stimulus
- A stimulus is sent by a ***WakeupCondition*** object
- Some wakeup classes:
  - **WakeupOnCollisionEntry**
  - **WakeupOnCollisionExit**
  - **WakeupOnCollisionMovement**
  - **WakeupOnElapsedFrames**
  - **WakeupOnElapsedTime**
  - **WakeupOnSensorEntry**
  - **WakeupOnSensorExit**
  - **WakeupOnViewPlatformEntry**
  - **WakeupOnViewPlatformExit**

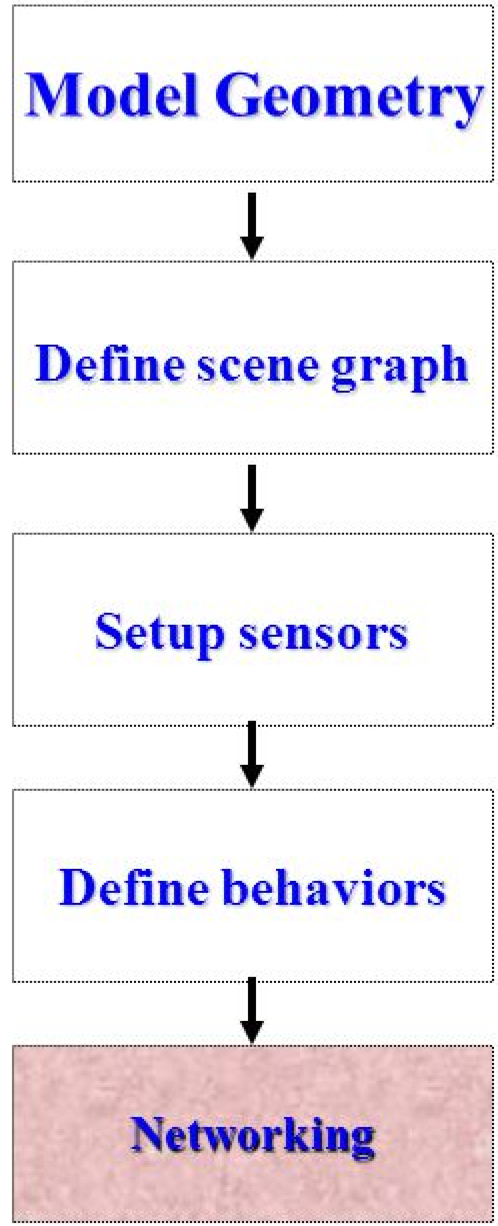
Java3D - Behavior usage

- We define a behavior Bhv that rotates the sphere by 1 degree
- We want this behavior to be called each frame so that the sphere will be spinning

```
WakeupOnElapsedFrames Wup = new WakeupOnElapsedFrames(0);
Bhv.wakeupOn(Wup);
```

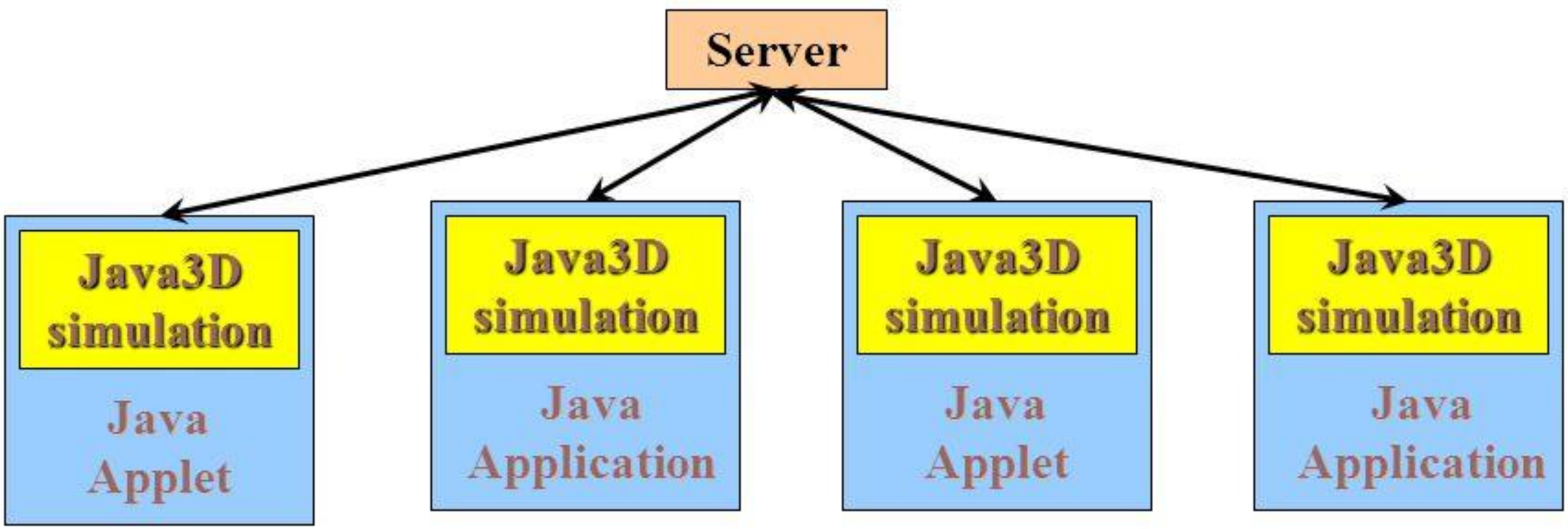
VC 6.4 on book CD

Java3D Initiation



Java3D - Networking

- Java3D does not provide a built-in solution for networked virtual environments
- However, it's perfect integration in the Java language allows the developer to use the powerful network features offered by Java
- Java3D applications can run as stand alone applications or as applets in a web browser



Java3D and VRML

- VRML provides possibilities for defining the objects and animating the objects in a virtual world
- Graphics APIs such as Java3D or WTK load from a VRML file only the static information, ignoring the sensors, routes, scripts, etc.
- Java3D structure is general enough to make the import of sensors and routes possible but currently we are not aware of any loader that does it
- One of the most popular library of Java3D loaders is the NCSA Portfolio (http://www.ncsa.uiuc.edu/~srp/Java3D/portfolio/)

NCSA Portfolio

Offers loaders for several model files

- 3D Studio (3DS)
- TrueSpace COB loader (COB)
- Java 3D Digital Elevation Map (DEM)
- AutoCAD (DXF)
- Imagine (IOB)
- Lightwave (LWS)
- Sense8 (NFF)
- Wavefront (OBJ)
- Protein Data Bank (PDB)
- Visualization Toolkit (VTK)
- VRML97

Loades the following parts of VRML97 files

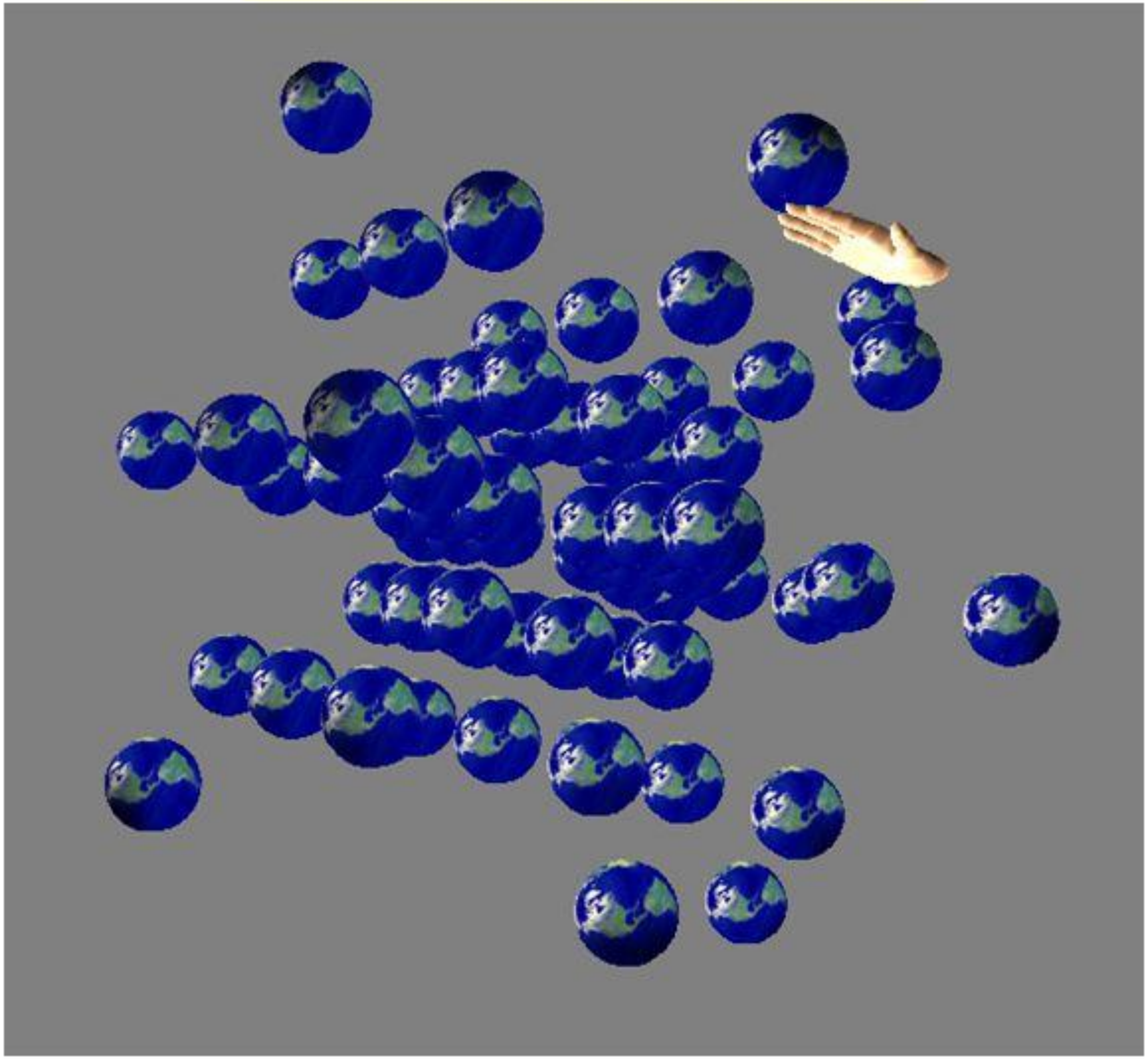
- Appearance
- Box
- Coordinate
- Collision (for grouping only)
- Group
- IndexedFaceSet
- IndexedLineSet
- Material
- Normal
- Shape
- Sphere
- Transform

Comparison between Java3D and WTK

- A comparative study was done at Rutgers between Java3d (Version 1.3beta 1) and WTK (Release 9);
- The simulation ran on a dual Pentium III 933 MHz PC (Dell) with 512 Mbytes RAM, with an Wildcat 4110 graphics accelerator which had 64 Mbytes RAM;
- The I/O interfaces were a Polhemus Insidetrack or the Rutgers Master II force feedback glove;
- The scene consisted of several 420-polygon spheres and a virtual hand with 2,270 polygons;
- The spheres rotated constantly around an arbitrary axis, while the hand was either rotating, or driven by the user.

Java3D –WTK Comparison

Graphics scene used in experiments

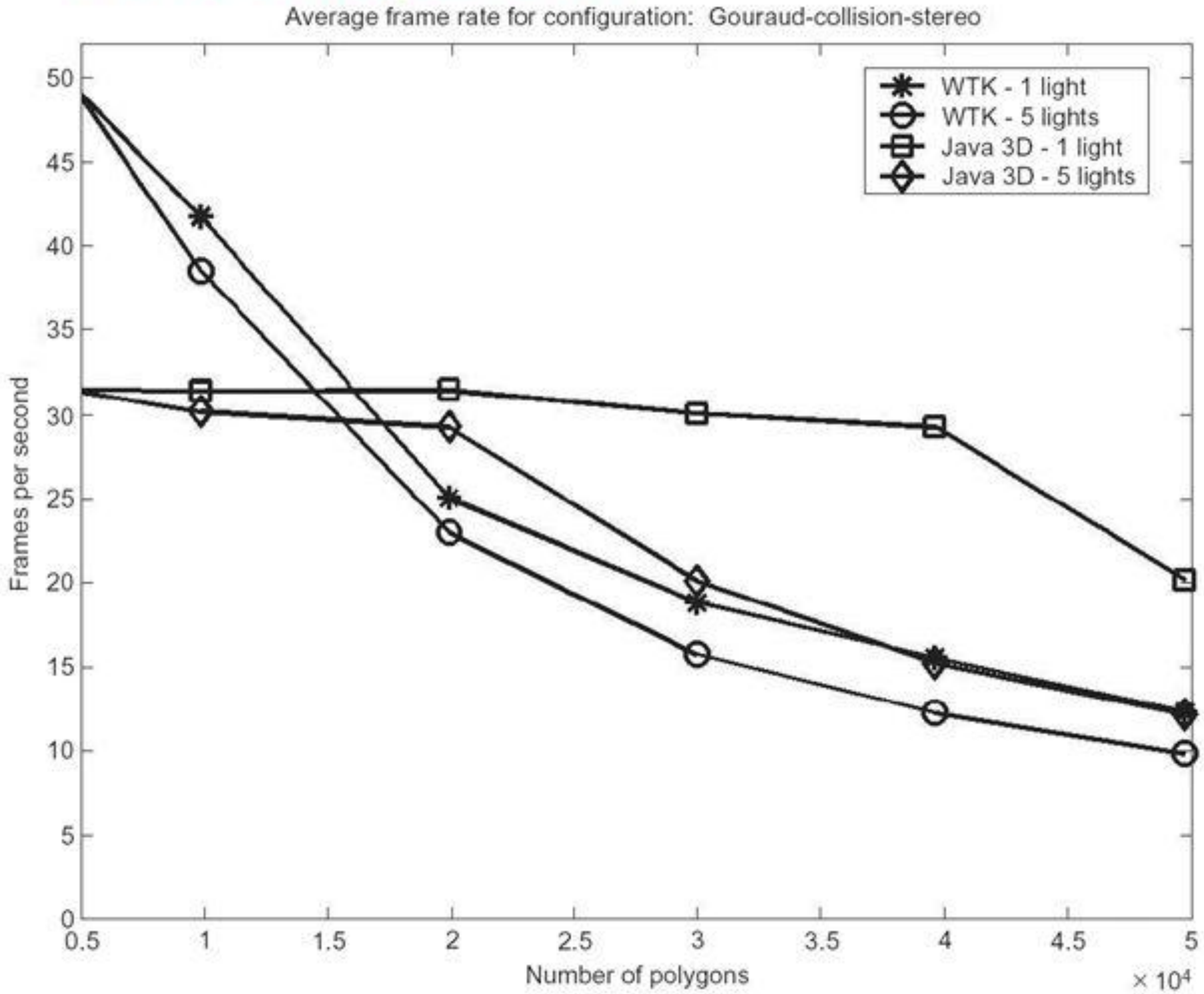


Comparison between Java3D and WTK

✓The simulation variables used to judged performance were:

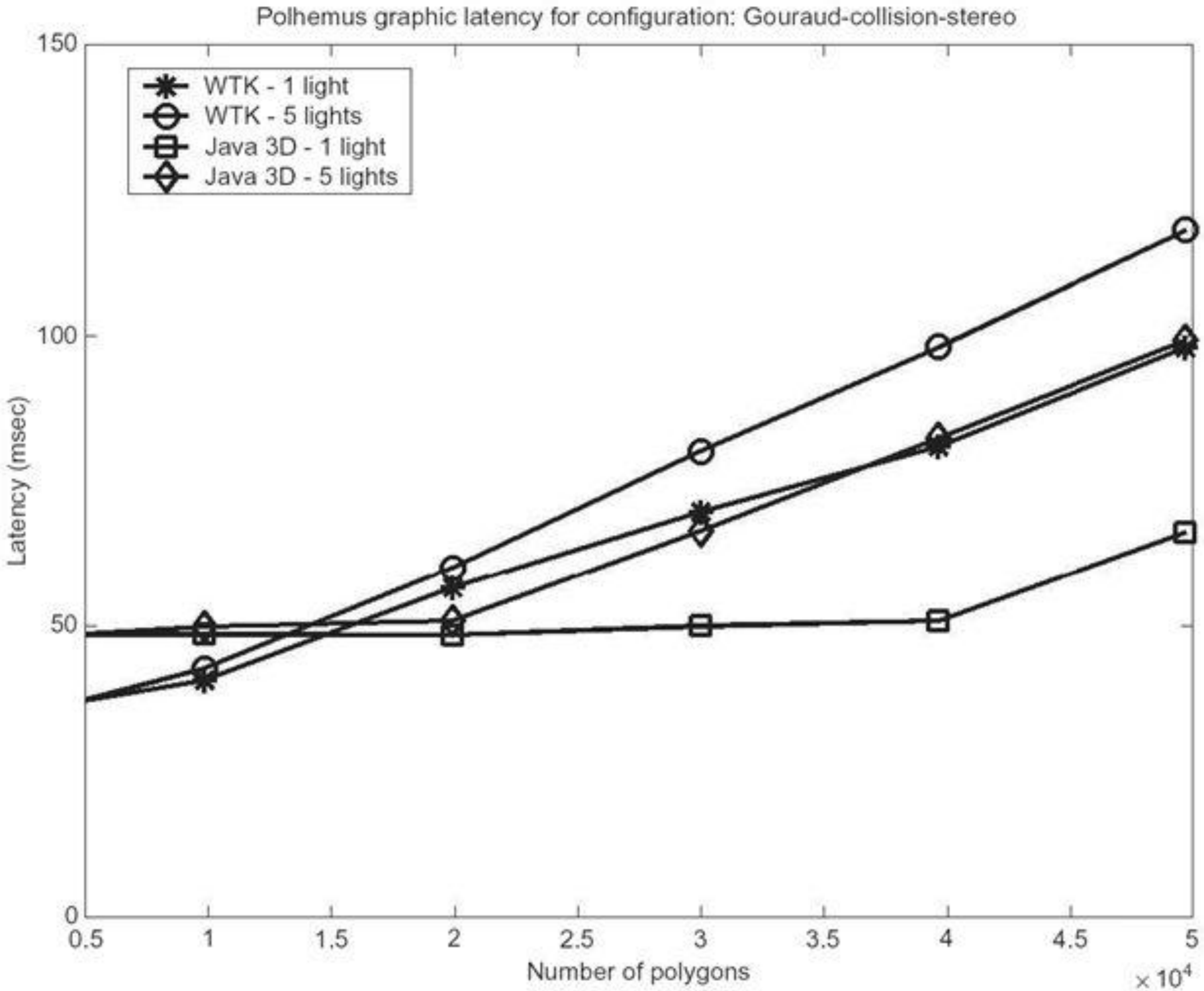
- graphic mode (monoscopic, stereoscopic),
- rendering mode (wireframe, Gouraud, textured);
- scene complexity (number of polygons 5,000 – 50,000);
- lighting (number of light sources 1, 5, 10);
- interactivity (no interaction, hand input, force feedback)

Java3D –WTK Comparison

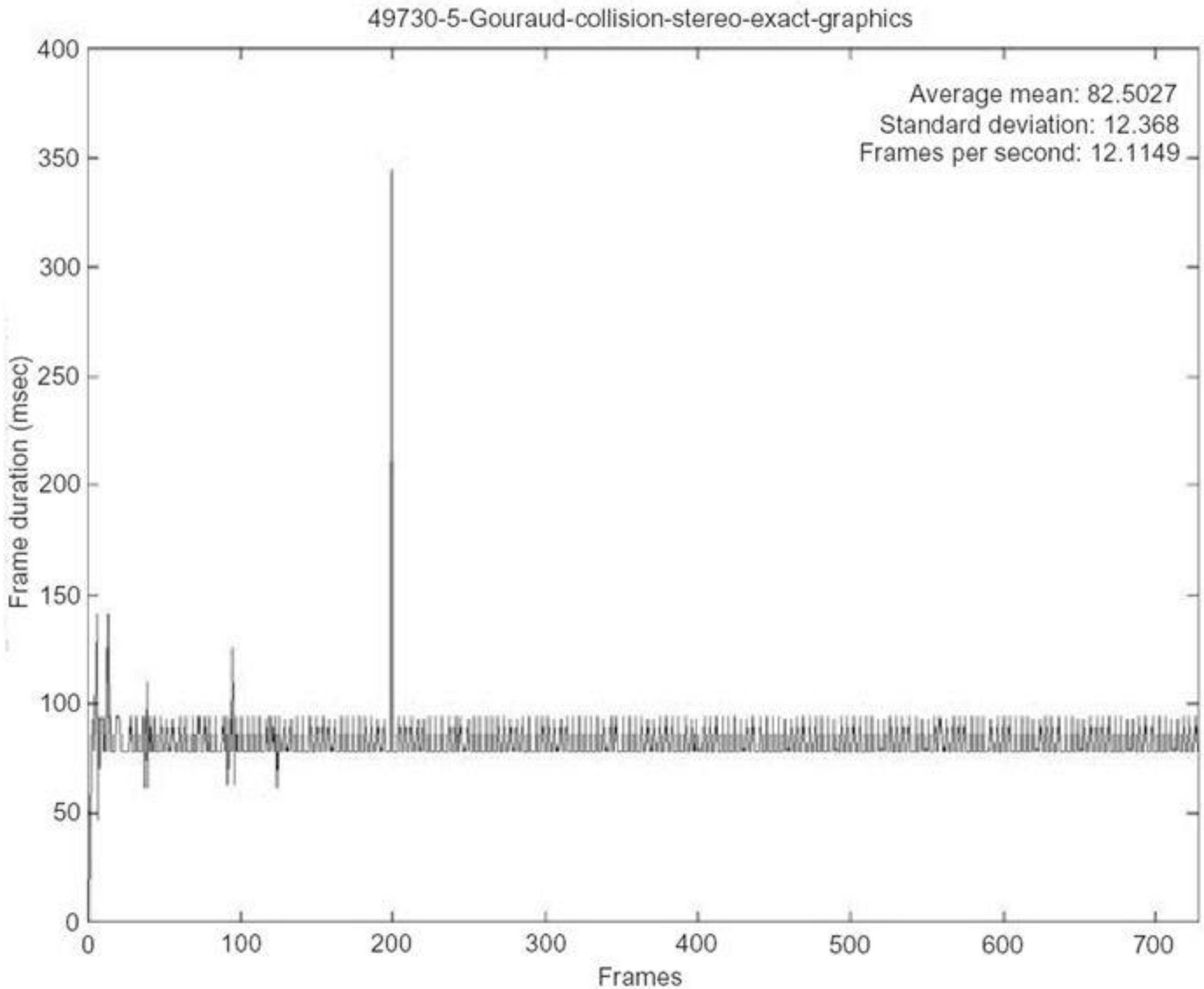


Java3d is faster on average than WTK, but has higher variability

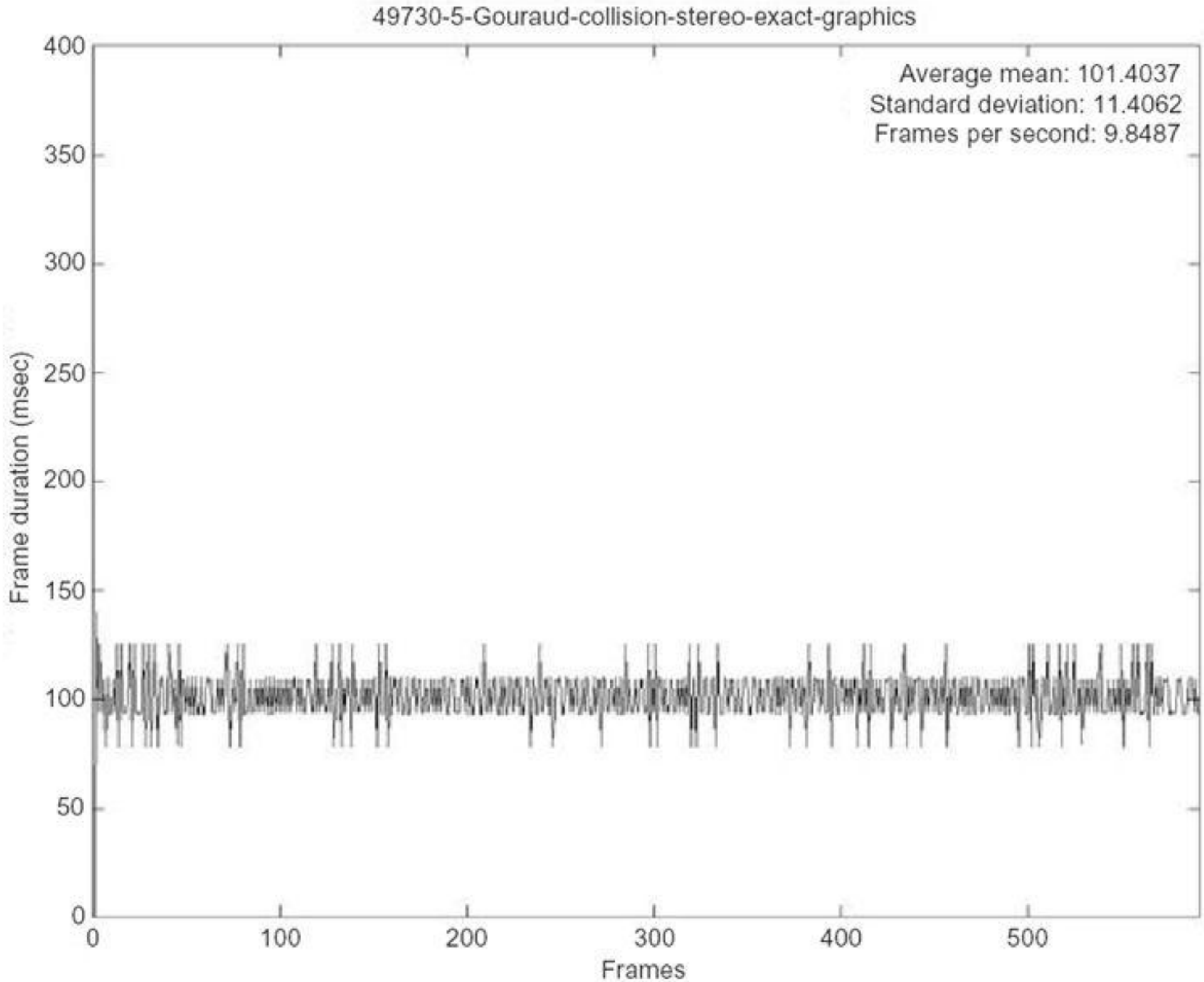
Java3D –WTK Comparison



Java3d Release 3.1 Beta 1 has less system latencies than WTK Release 9



But Java3d has more *variability* in the scene rendering time



WTK does not have spikes in the scene rendering time

Thank You