Practical Data Visualization and Virtual Reality

Virtual Reality
VR Software and Programming

Karljohan Lundin Palmerius
Synopsis

- Scene graphs
- Event systems
- Multi screen output and synchronization
- VR software systems
Scene Graphs

- Directed Acyclic Graph (DAG)
  - Hierarchy of nodes (tree)
  - Reflects hierarchy of objects
  - Traversed for processing

- Essentials
  - Nodes (group or leaf)
  - Edges

- Nodes
  - Encapsulate behaviour
  - E.g. group, geometry, properties
Scene Graph

- Structure
  - Data structure / data exchange formats
  - Standards (VRML, X3D, MPEG-4, etc)
- Software
  - OpenInventor
  - H3D API
  - OpenSceneGraph (OSG)
  - OpenSG
Scene Graph

- Program package
  - (Partly) replaces graphics API
  - Should not hinder
    ...only help

- Framework
  - Extend with new functionality
Scene Graphs Provide

- High performance rendering
- Structured programming
  - Less effort
  - Logical structure
  - Code reuse
- General purpose, dynamic structure
  - Rapid prototyping
  - Real-time updates
  - Interaction and modification
Software Provides

- Wide selection
  - Primitives, materials, lights
- Media import
- Various manipulations
  - Interactors, camera settings
- Events, timers, scripting, etc.
  etc...
VRML/X3D Style

- Transforms are groups
  - Implicit separation
- Shape nodes
  - Selected properties
  - Geometry child
  - No leaking property
Node Reuse

- More than one parent
  - Allowed by DAG
  - Share child node
  - Less memory use
  - Less code
  - Less to update
VRML/X3D Molecule Example

- Water Molecule
  - 1 Oxygen
  - 2 Hydrogen
X3D Code

• **Tree structure**
  - Hierarchy reflected in XML structure
  - XML nodes become scene graph nodes

• **Node children are specified as node contents**
  - E.g. `<Group><Group/></Group>`
  - Default field name for all nodes
  - Explicitly define container
    e.g. `containerField="xyz"`

• **Values are specified as attributes**
  - E.g. `<Material diffuseColor="0.9 0.4 0.6"/>`
<Root>
  <Transform translation="432.2 -923.6 12.4"
            rotation="0.257 -0.950 -0.181 1.175"
            scale="0.2 0.2 0.2">
    <Transform DEF="OXYGEN"
               translation="0 0 0">
      <Shape>
        <Appearance><Material diffuseColor="0.9 0.9 0.9"/></Appearance>
        <Sphere radius="0.2"/>
      </Shape>
    </Transform>
  </Transform>
  <Transform DEF="HYDROGEN_1"
            translation="0.3 0.3 0">
    <Shape DEF="H_SHAPE">
      <Appearance><Material diffuseColor="0.8 0.3 0.3"/></Appearance>
      <Sphere radius="0.1"/>
    </Shape>
  </Transform>
  <Transform DEF="HYDROGEN_2"
            translation="0.3 -0.3 0">
    <Shape USE="H_SHAPE"/>
  </Transform>
</Root>
OpenScenegraph C++

• Explicit definition
  – Create every node
  – Specify children
  – Set every non-default property

• Import model from file
  – Put model into scene graph
  – Modifying properties
    • Manual traversal
    • Automatic by visitors
int main(){
    osgViewer::Viewer viewer;

    osg::PositionAttitudeTransform* moleculeXForm
        = new osg::PositionAttitudeTransform;

    osg::PositionAttitudeTransform* oxygenXForm
        = new osg::PositionAttitudeTransform;
    osg::PositionAttitudeTransform* hydrogen1XForm
        = new osg::PositionAttitudeTransform;
    osg::PositionAttitudeTransform* hydrogen2XForm
        = new osg::PositionAttitudeTransform;

    osg::Geode* oxygenGeode = new osg::Geode;
    osg::Geode* hydrogen1Geode = new osg::Geode;
    osg::Geode* hydrogen2Geode = new osg::Geode;

    osg::Sphere O = new osg::Sphere();
    osg::ShapeDrawable* shapeO = new osg::ShapeDrawable(O);

    osg::Sphere H1 = new osg::Sphere();
    osg::ShapeDrawable* shapeH1 = new osg::ShapeDrawable(H1);

    osg::Sphere H2 = new osg::Sphere();
    osg::ShapeDrawable* shapeH2 = new osg::ShapeDrawable(H2);
```cpp
oxygenGeode->addDrawable(shapeO);
hydrogen1Geode->addDrawable(shapeH1);
hydrogen2Geode->addDrawable(shapeH2);

oxygenXForm->addChild(oxygenGeode);
hydrogen1XForm->addChild(hydrogen1Geode);
hydrogen2XForm->addChild(hydrogen2Geode);

moleculeXForm->addChild(oxygenXForm);
moleculeXForm->addChild(hydrogen1XForm);
moleculeXForm->addChild(hydrogen2XForm);

viewer.setSceneData(moleculeXForm);
return viewer.run();
```
Networked Objects
Event Systems
Event Systems

• Networked Objects
  – Define data flow
  – Automated run-time processing

• Software
  – H3D API
  – Visualization Toolkit (VTK)
  – Voreen (Volume Rendering Engine)
  – Qt
  – etc...
H3D API

- Networked scene graph
- Events in the scene graph
  - More tasks given to scene graph
  - Program distributed as "engines"
- Connecting fields
  - Sending values
  - Event network
Event Systems

- **Nodes**
  - Encapsulate a behaviour
- **Fields**
  - Define properties of nodes
  - Contain data: colour, translation, light intensity, etc
- **Sensors**
  - Event sources: Timers, triggers, mouse, etc
- **Engines**
  - Activated by incoming events
  - Process incoming data and computes new data
Event Systems

• Fields
  – Contains values, e.g., SField, MField
    • Copy values, calculate new values
  – Contains nodes, e.g., SFNode, MFNode
    • Copy pointers, create new objects
    • Require good reference counters

• Routing
  – One field value to many fields
  – Engine may accept many input
  – Lazy evaluation
Simple Example

- 3D mouse sensor
  - Controls object transform
    - SFVec3f rawTranslation
    - SFVec3f accumulatedTranslation
    - SFRotation rawRotation
    - SFRotation accumulatedRotation
    - SFInt32 buttons
  - Connect field
    - route *from* 3D mouse translation field
    - route *to* X-form translation field
Engine Example

- PythonScript node
  - Field type implemented as class (F2I)
  - Field instance defined as class instance (float2int)
  - F2I float2int
Event Script Example

- Animation
  - Switch to select "frame"
  - Floating point TimeSensor
  - Script to convert float to integer
from H3D import *
from H3DInterface import *

class F2I( TypedField( SFInt32, SFFloat ) ):
    def update( self, event ):
        return int(event.getValue())

float2int = F2I()
<?xml version="1.0" encoding="UTF-8"?>
<Group>
  <Switch DEF="VSWITCH"
    whichChoice="0">
    ...
  </Switch>
</Group>
<TimeSensor DEF="TIME"
cycleInterval="5"
startTime="0"
loop="true" />

<ScalarInterpolator DEF="FRAMES"
key="0 0.2 0.4 0.6 0.8 1"
keyValue="0 1 2 3 4 5" />

<ROUTE fromNode="TIME" fromField="fraction_changed"
toNode="FRAMES" toField="set_fraction" />

<PythonScript DEF="PY_FRAME"
url="x3d/setup_animation.py" />

<ROUTE fromNode="FRAMES" fromField="value_changed"
toNode="PY_FRAME" toField="float2int" />
<ROUTE fromNode="PY_FRAME" fromField="float2int"
toNode="VSWITCH" toField="whichChoice" />

</Group>
VR Software Systems
Multi Screen Output

Application

Scene Graph

VR Software

OpenGL

Hardware

Display
Multi-processing

- **Threads**
  - Single computer
  - Single multi-head graphics card
  - Single process with one memory address space

- **Processes**
  - Single computer
  - One process per graphics card and display

- **Clusters**
  - Multiple computers
  - One graphics card per computer
  - Separate memory
Memory and State Access

- State synchronization and locking
  - Necessary for consistent rendering
  - To avoid tearing

- Techniques
  - Threads – share memory
  - Processes – explicit ”shared memory”
  - Serialization over network
    - e.g. TCP/IP
    - Low latency is more important than bandwidth
    - One master, many slaves
Synchronization

• Needs for synchronization
  • avoid inconsistent behaviour across the screens

• Types
  – state synchronization
    • what states to render
  – frame synchronization
    • when to switch to the next rendered frame
    • software swap lock over TCP/IP
  – generator lock (gen-lock hardware)
    • when to send the front buffer to the display
    • when to draw left/right image
VR Software Systems

• Purpose
  • set up cluster nodes and start the VR software
  • handle per display settings, such as viewpoint
  • handle state serialization and synchronization
  • connect to tracking systems, audio, etc.

• Approach
  • typically a framework architecture
  • VR System handles the thread
  • provides callbacks for processing and rendering
VR Software Systems

- **VRJuggler**
  - Comprehensive
  - Advanced
  - Complex

- **SGCT**
  - Simple
  - In-house
#include <sgct.h>

void myInitFun();
void myDrawFun();
void myPreSyncFun();
void myCleanUpFun();

int main( int argc, char* argv[] )
{
    sgct::Engine gEngine( argc, argv );

    gEngine.setInitOGLFunction( myInitFun );
gEngine.setDrawFunction( myDrawFun );
gEngine.setPreSyncFunction( myPreSyncFun );
gEngine.setCleanUpFunction( myCleanUpFun );

    if( !gEngine.init( sgct::Engine::OpenGL_3_3_Core_Profile ) ){
        return EXIT_FAILURE;
    }

    gEngine.render();

    exit( EXIT_SUCCESS );
}
SGCT State Synchronization

- Encode/decode callback

```cpp
goct::SharedDouble angle(0.0);
goct::SharedDouble distance(0.0);
...

int main( int argc, char* argv[] )
{
    gEngine = new goct::Engine( argc, argv );
    ...
    goct::SharedData::instance() -> setEncodeFunction( myEncodeFun );
    goct::SharedData::instance() -> setDecodeFunction( myDecodeFun );
    ...
}

void myEncodeFun()
{
    goct::SharedData::instance() -> writeDouble( &angle );
    goct::SharedData::instance() -> writeDouble( &distance );
}

void myDecodeFun()
{
    goct::SharedData::instance() -> readDouble( &angle );
    goct::SharedData::instance() -> readDouble( &distance );
}
```