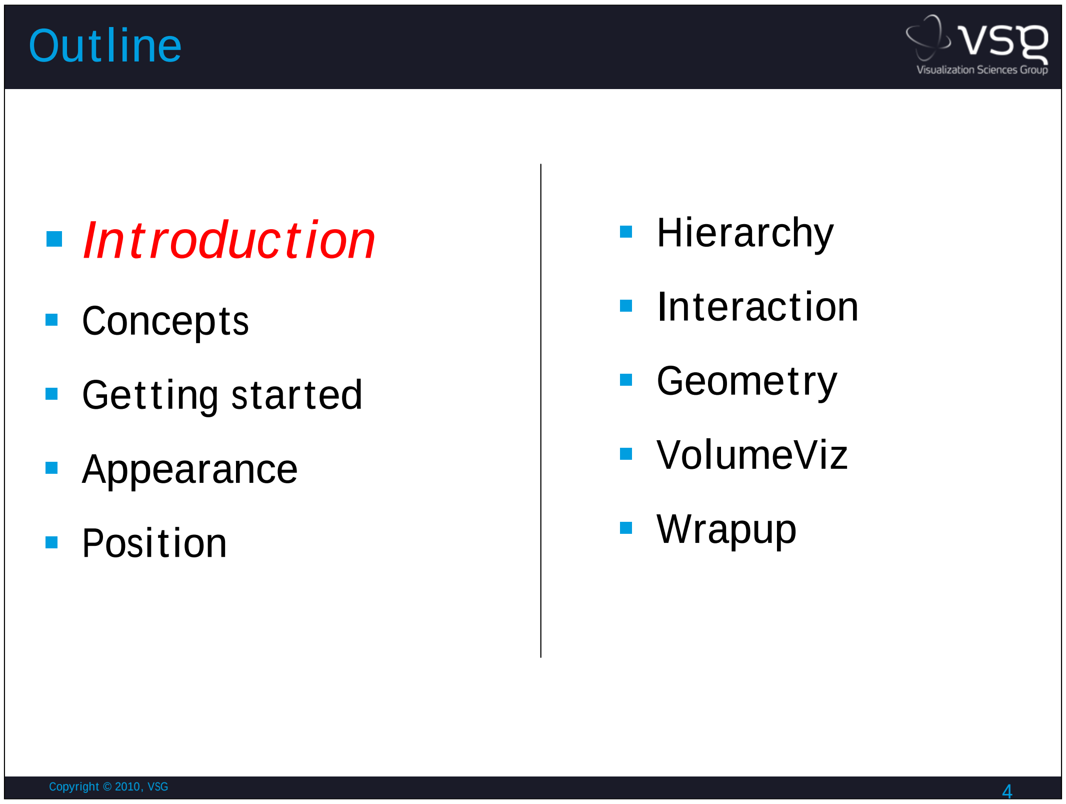
Open Inventor by VSG *Fast Start*

Mike Heck
Technology Advisor
February 2010

Welcome to the Open Inventor by VSG "Fast Start" webinar. This is a short, but very concentrated, introduction to programming with the Open Inventor toolkit for 3D visualization. It's not a substitute for taking a full Open Inventor training class or even for reading the more complete tutorials and Users Guide. There are a lot of powerful features in Open Inventor that you may want to use, but we just won't have time to discuss in this presentation. However we will try to list some of the features that you may want to read further about.



Outline

- *Introduction*
- Concepts
- Getting started
- Appearance
- Position
- Hierarchy
- Interaction
- Geometry
- VolumeViz
- Wrapup

So we won't be able to teach you everything in Open Inventor today, but if you need to get started in a hurry and get some data up on the screen RIGHT NOW, then by the end of this presentation you should have a good idea of:

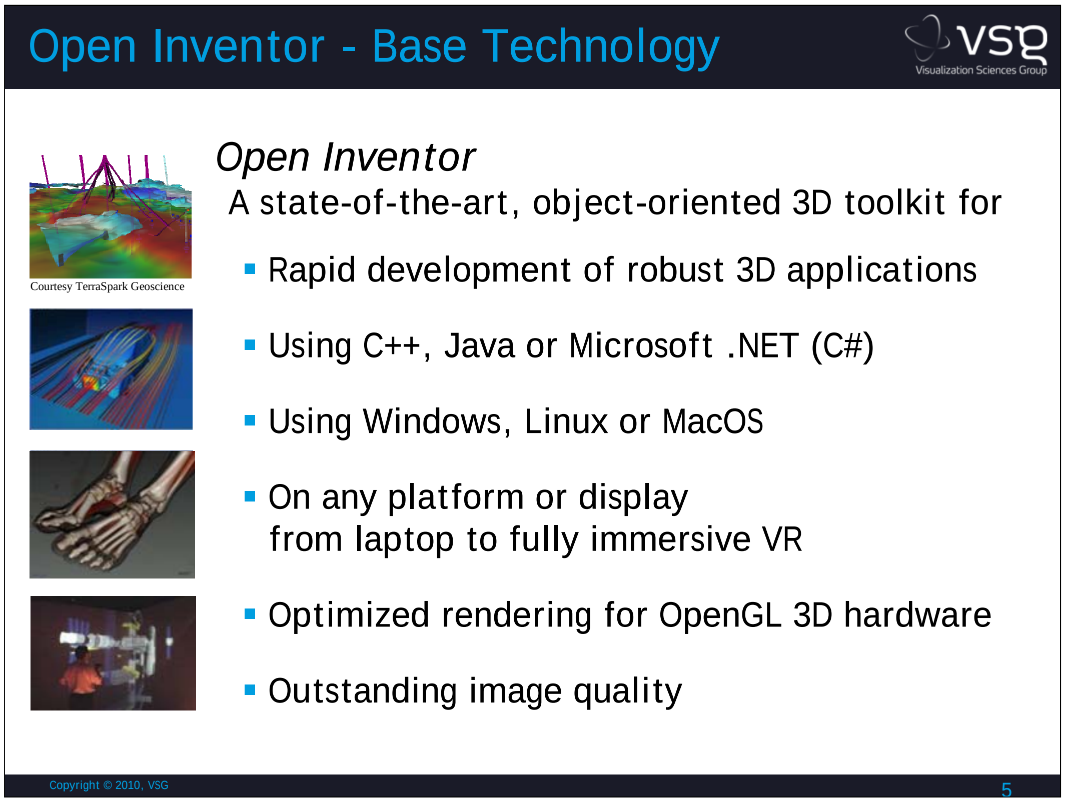
•Core Open Inventor concepts that you will use over and over

•How to build, view and modify a 3D scene

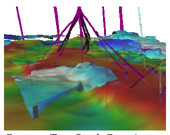
•How to create 3D geometry like lines, polygons and text

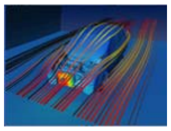
•How to load and display 3D volume data, like medical scans, seismic data, etc.

In fact by the end of this presentation we will have created a basic, functional Open Inventor program that loads a data file, renders some geometry and allows the user to interactively modify the scene. Even if you're only thinking about programming with Open Inventor, or you just want to get an idea of what programming with Open Inventor is like, we hope you will be impressed with both the power and the elegant design of this API.



Open Inventor - Base Technology


Courtesy TerraSpark Geoscience

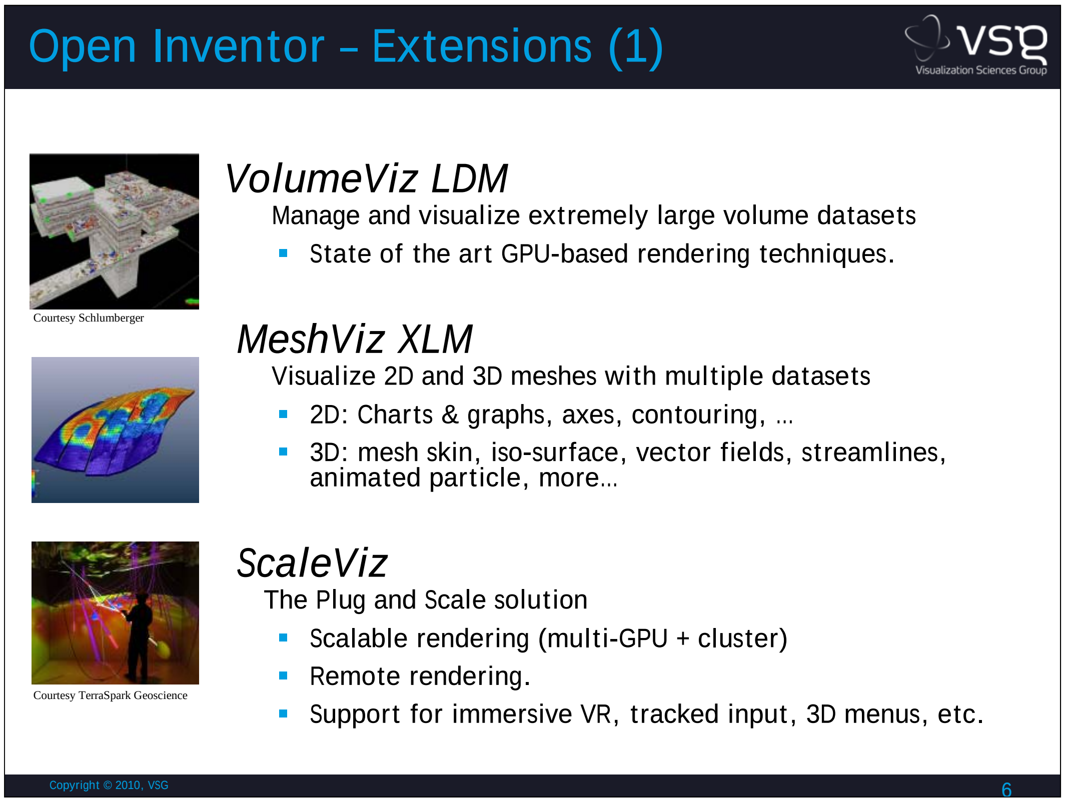


Open Inventor

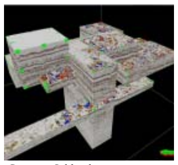
A state-of-the-art, object-oriented 3D toolkit for

- Rapid development of robust 3D applications
- Using C++, Java or Microsoft .NET (C#)
- Using Windows, Linux or MacOS
- On any platform or display
  from laptop to fully immersive VR
- Optimized rendering for OpenGL 3D hardware
- Outstanding image quality

Open Inventor is our core 3D visualization library. An amazing variety of 3D applications are built with Open Inventor, including, as you see here, geoscience applications, engineering design and simulation applications, medical applications and virtual reality applications. In addition to portability, Open Inventor will save you a lot of time in development of your application both because you work at a higher level than OpenGL and because many operations that 3D developers typically implement, like moving the camera and selecting objects, are already implemented for you in Open Inventor.



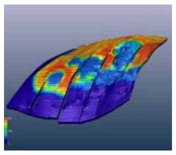
Open Inventor – Extensions (1)


Courtesy Schlumberger

VolumeViz LDM

Manage and visualize extremely large volume datasets

- State of the art GPU-based rendering techniques.



MeshViz XLM

Visualize 2D and 3D meshes with multiple datasets

- 2D: Charts & graphs, axes, contouring, …
- 3D: mesh skin, iso-surface, vector fields, streamlines, animated particle, more…

Courtesy TerraSpark Geoscience

ScaleViz

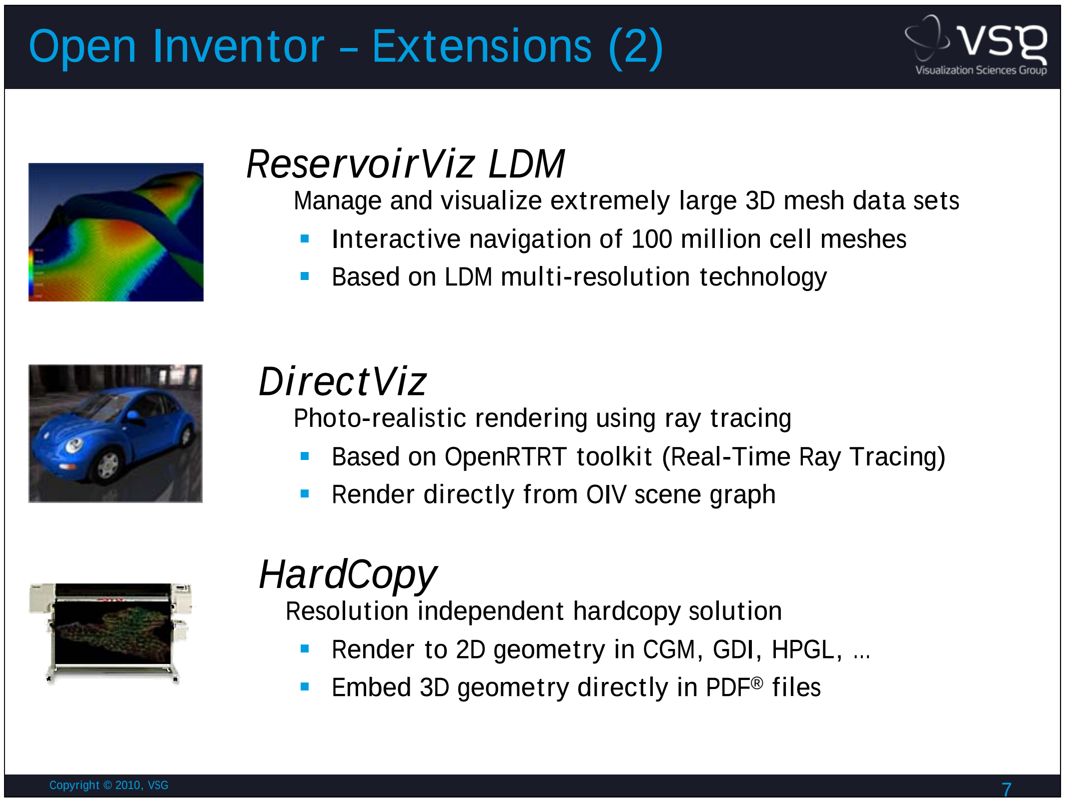
The Plug and Scale solution

- Scalable rendering (multi-GPU + cluster)
- Remote rendering.
- Support for immersive VR, tracked input, 3D menus, etc.

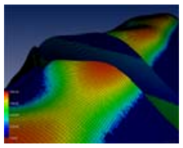
We've also built a number of "extensions". They actually extend the set of objects, data types and rendering techniques that are pre-built for you. Let's review them quickly so you'll have an idea which extensions will help you with your application. VolumeViz, which we will talk about later today, is a full featured toolkit for managing and visualizing volume data, whether it's a stack of images from microscopy, a CT scan or a seismic survey. VolumeViz LDM provides data management that allows applications to interactively render data far larger than available system memory.

<CLICK!> MeshViz adds support for managing, extracting and visualizing mesh data, whether it's a finite element analysis, computational fluid dynamics or a petroleum reservoir. MeshViz XLM provides a unique "interface" based API that allows applications to visualize any type of mesh with any type of cells and without copying any application data.

<CLICK!> ScaleViz allows your application to scale up the number of displays, the data capacity and/or the rendering performance by adding additional GPUs and/or workstations. In addition it provides support for immersive virtual reality and tracked input.



Open Inventor – Extensions (2)



ReservoirViz LDM

Manage and visualize extremely large 3D mesh data sets

- Interactive navigation of 100 million cell meshes
- Based on LDM multi-resolution technology

DirectViz

Photo-realistic rendering using ray tracing

- Based on OpenRTRT toolkit (Real-Time Ray Tracing)
- Render directly from OIV scene graph

HardCopy

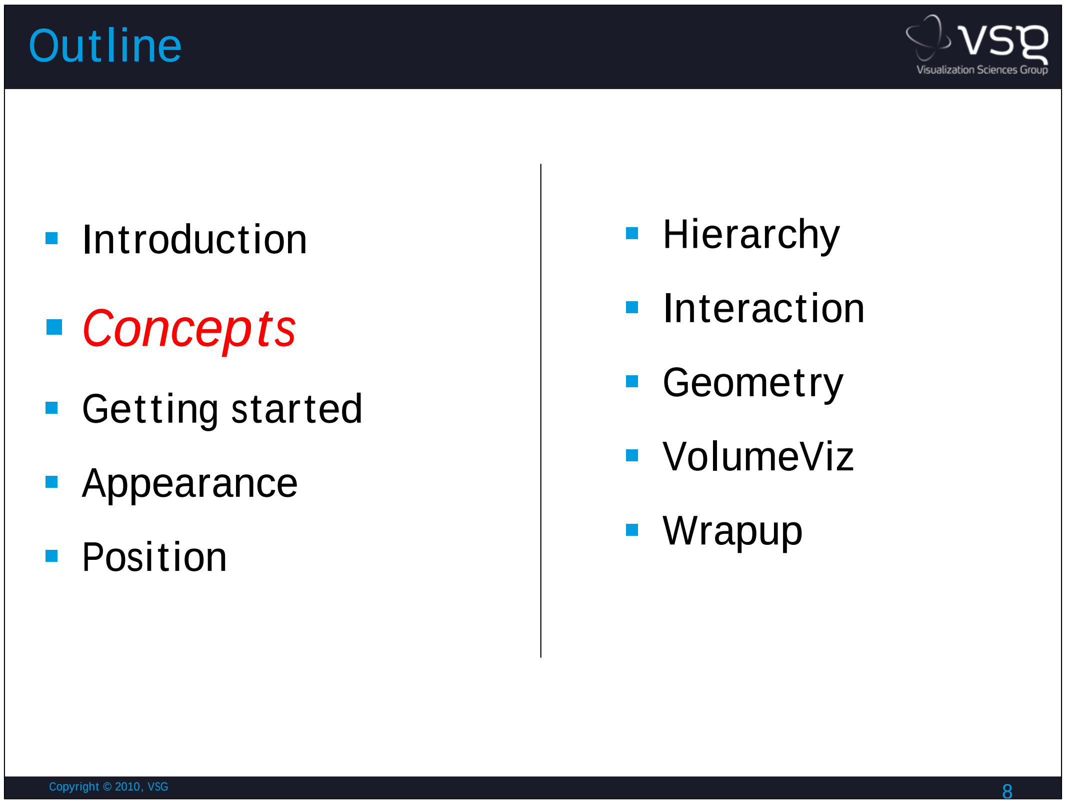
Resolution independent hardcopy solution

- Render to 2D geometry in CGM, GDI, HPGL, …
- Embed 3D geometry directly in PDF® files

Copyright © 2010, VSG 7

ReservoirViz LDM is a combination of mesh visualization with the LDM data management engine to allow interactive navigation of very large petroleum reservoir meshes.
The DirectViz extension allows you to render your 3D scene with photo-realistic quality using our real-time ray tracing engine OpenRTRT. You use the same 3D scene description for GPU rendering and for ray traced rendering. You can also use the toolkit for ray tracing based computation.

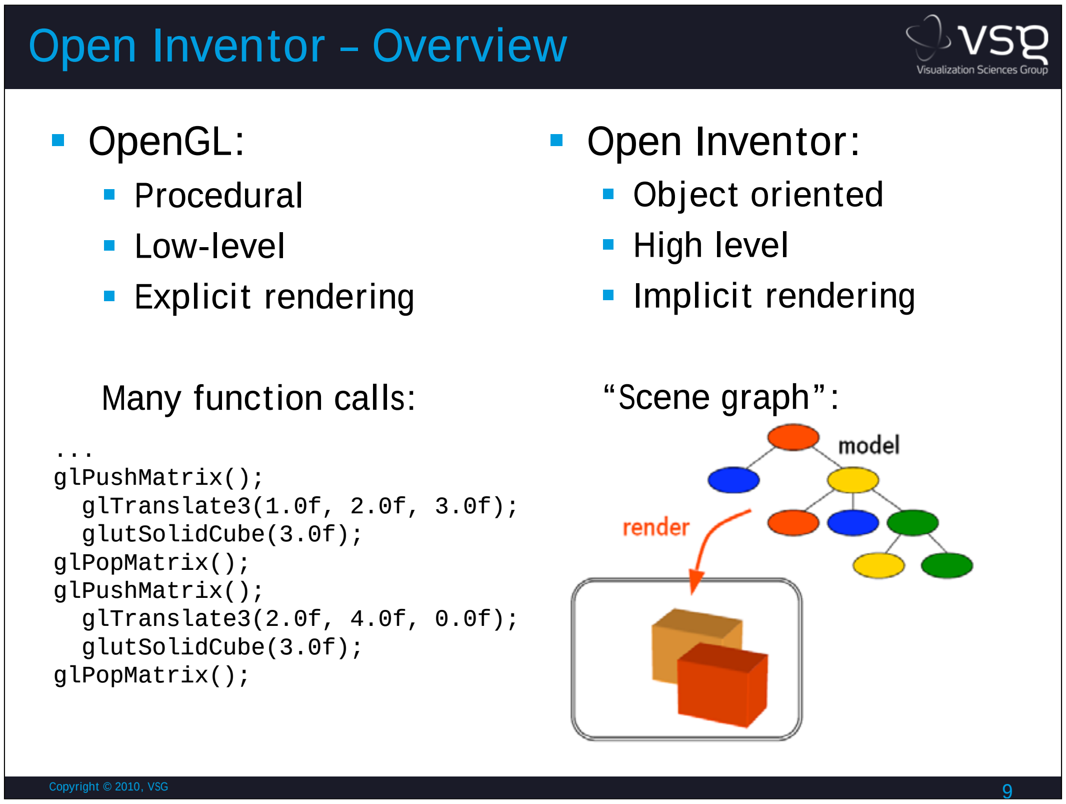
The HardCopy extension provides two solutions. First it can convert a view of your 3D scene into a set of 2D primitives (with hidden surfaces removed of course) that can be, for example, sent a large format plotter much more efficiently than a very large image. Second it can convert your 3D scene into a form that can be directly embedded in an Adobe PDF file and viewed using the Acrobat Reader's built-in 3D viewer.



Outline

- Introduction
- *Concepts*
- Getting started
- Appearance
- Position
- Hierarchy
- Interaction
- Geometry
- VolumeViz
- Wrapup

OK, let’s get started learning about Open Inventor.



Open Inventor – Overview

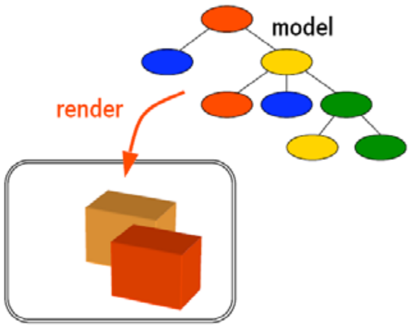
- OpenGL:
  - Procedural
  - Low-level
  - Explicit rendering

Many function calls:

```
...
glPushMatrix();
  glTranslate3(1.0f, 2.0f, 3.0f);
  glutSolidCube(3.0f);
glPopMatrix();
glPushMatrix();
  glTranslate3(2.0f, 4.0f, 0.0f);
  glutSolidCube(3.0f);
glPopMatrix();
```

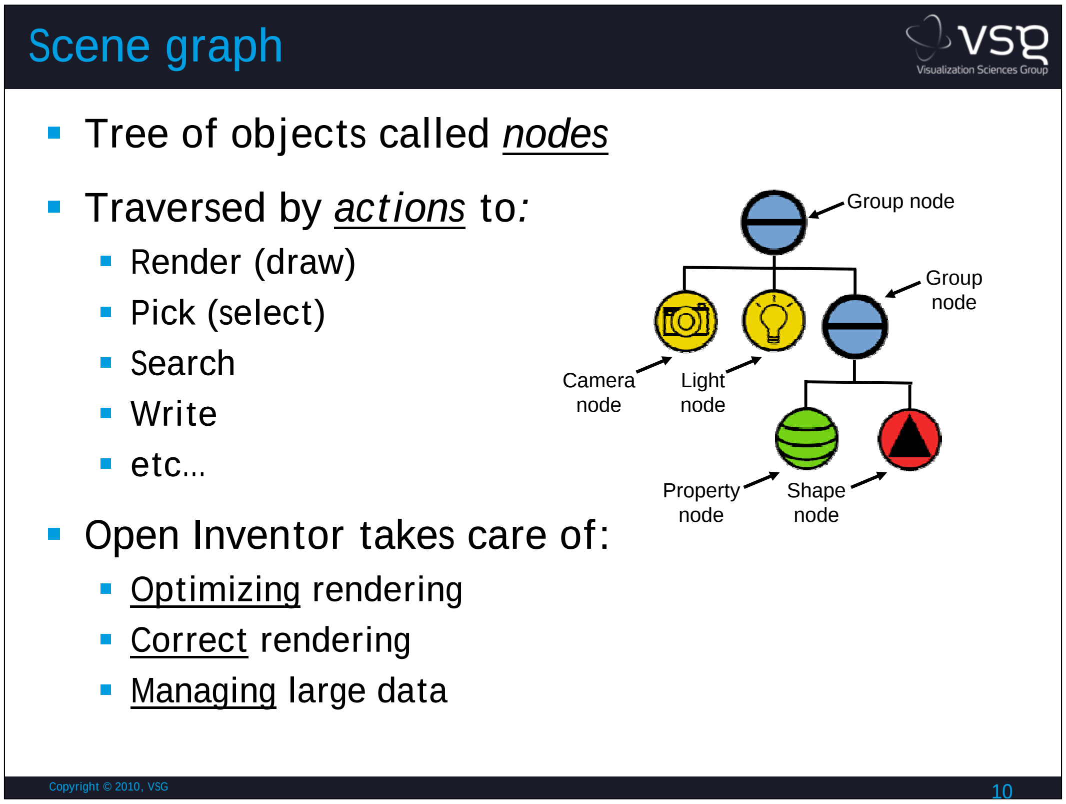
- Open Inventor:
  - Object oriented
  - High level
  - Implicit rendering

“Scene graph”:



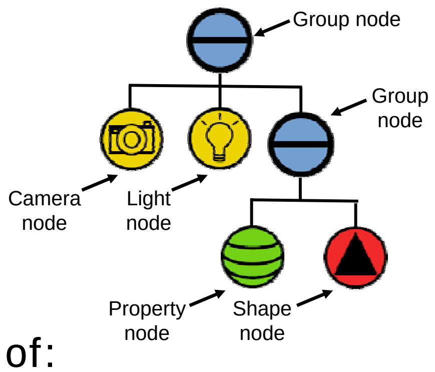
Open Inventor uses OpenGL, the industry standard hardware interface API, for rendering. OpenGL is sometimes used directly for application development, so at a high level, let’s contrast OpenGL development with Open Inventor development. Rendering a simple scene in OpenGL basically involves “C” programming and a lot of function calls. In fact we’ve hidden the actual rendering of the geometry by using a utility function from the open source GLUT library, but basically we have to tell OpenGL exactly how to render each primitive. Notice that in addition to the function calls we have to do “state” management explicitly, for example saving and restoring the current transformation so the cubes can be separately positioned.

<CLICK!> Open Inventor is a C++ (and Java and C#) library, so we work with “objects” that manage their own properties and allow us to simply modify those properties rather than making different rendering calls. We build a “tree” of these objects called the “scene graph” that describes what the 3D scene should look like, rather than exactly how to render it. Open Inventor takes care of the rendering for us. It also takes advantage of knowing the structure of the scene in advance to optimize the rendering. However the interaction of Open Inventor and OpenGL is carefully defined so it is possible to define new objects derived from existing ones and also, if necessary, to mix Open Inventor and OpenGL calls.



Scene graph

- Tree of objects called *nodes*
- Traversed by *actions* to:
  - Render (draw)
  - Pick (select)
  - Search
  - Write
  - etc…
- Open Inventor takes care of:
  - Optimizing rendering
  - Correct rendering
  - Managing large data



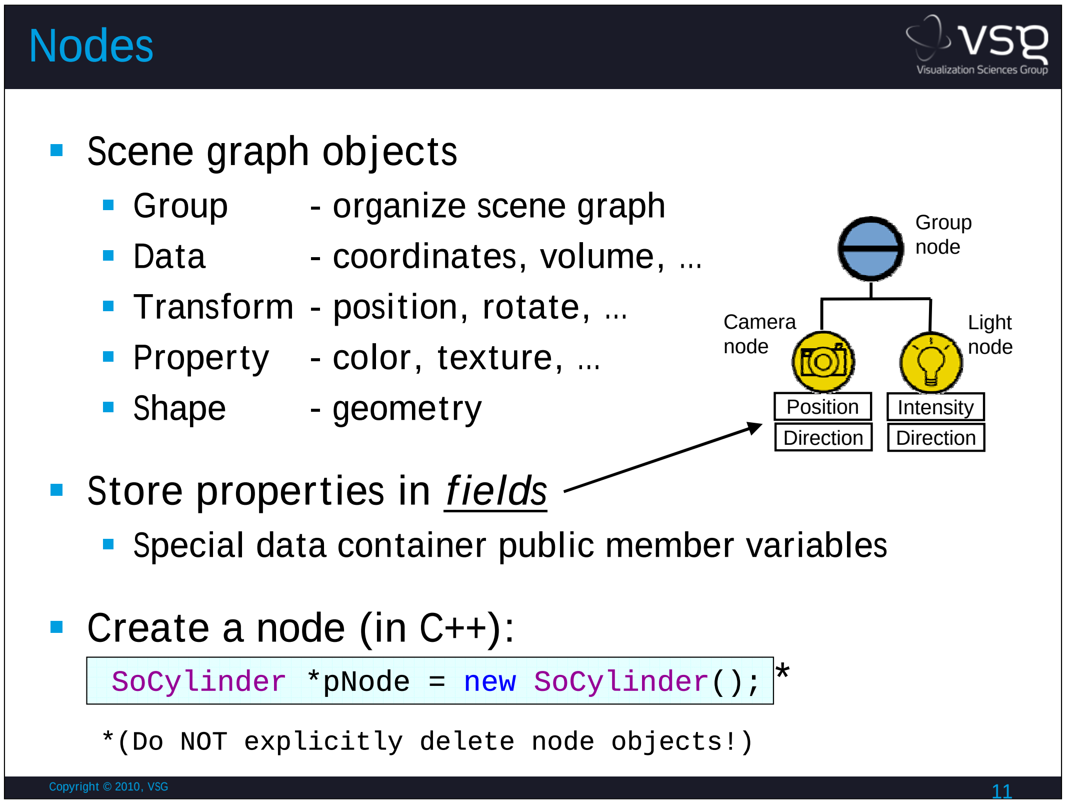
Copyright © 2010, VSG

10

As I said, the objects in the scene graph form a tree structure. We call these objects **nodes**. You can see an example of a simple scene graph here and some examples of the kinds of nodes we'll use to build the scene graph.

<CLICK!> Another kind of object, called an **action**, traverses or walks the scene graph, visiting each node in turn and triggering some appropriate action by the node. Examples of actions we'll use in Open Inventor are: Render - to draw the scene, Pick – to select geometry by pointing, and Search – to find a specific node, or nodes.

<CLICK!> In summary: Your application will be responsible for defining the appearance and the "meaning" of the 3D objects in the scene. This is the most important stuff, where you use your domain specific knowledge to create value for your customers. Open Inventor will be responsible for rendering an image of those objects, using our knowledge of OpenGL and graphics hardware to maximize performance and image quality and ensure complex scenes are rendered correctly.



Nodes

- Scene graph objects
  - Group - organize scene graph
  - Data - coordinates, volume, ...
  - Transform - position, rotate, ...
  - Property - color, texture, ...
  - Shape - geometry
- Store properties in *fields*
  - Special data container public member variables
- Create a node (in C++):

```
SoCylinder *pNode = new SoCylinder();
```
*

*(Do NOT explicitly delete node objects!)

Group node
Camera node
Light node
Position
Direction
Intensity
Direction

11

The general categories of scene graph objects, or nodes, are:

•Group nodes define the structure of the scene graph

•Data nodes hold (or point to) data like coordinates

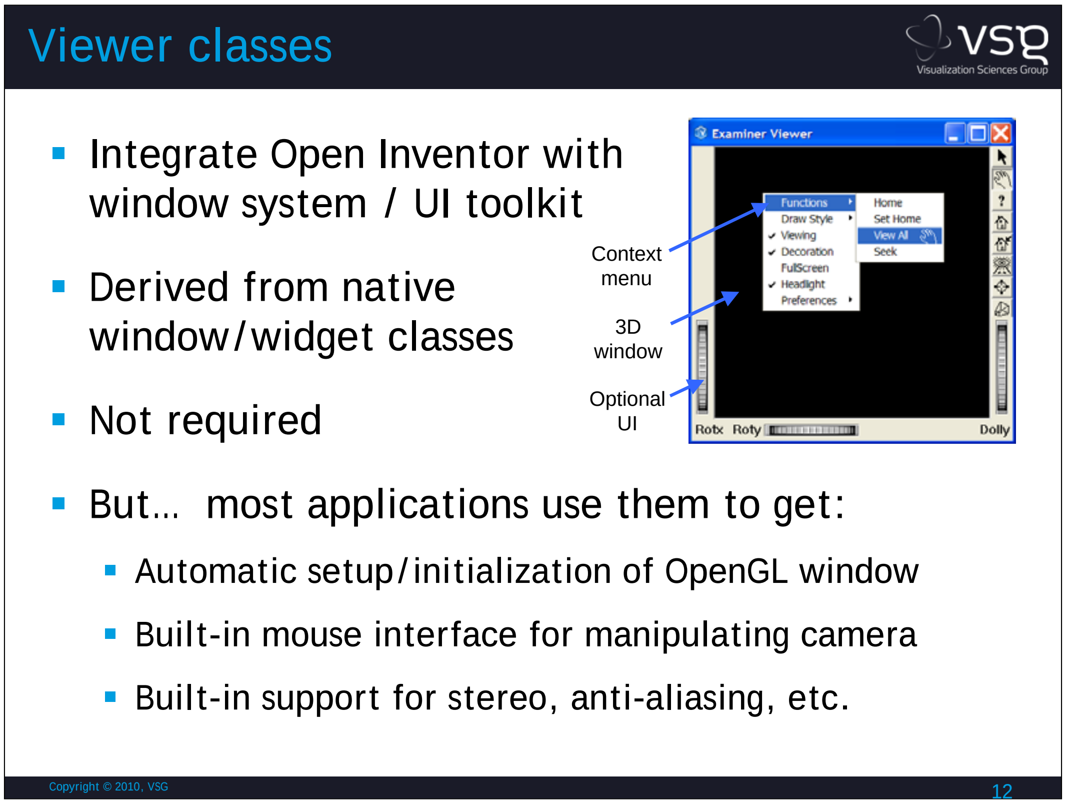
•Transform nodes scale, rotate and position geometry

•Property nodes define properties like color, font size, etc.

•Shape nodes define the actual geometry – using data, properties and transforms

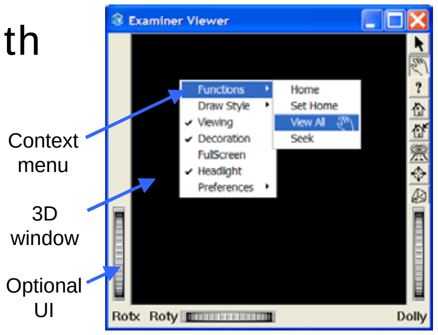
<CLICK!> All nodes store their properties, or state, in special member variables called **fields**. For example a camera node might have the properties “Position” and “Direction”, while a light source node might have the properties “Intensity” and “Direction”. We’ll talk more about fields and how to use them a little bit later. For now let’s actually create a node object.

<CLICK!> This is the C++ syntax, but in the other languages the class names are the same and the syntax is very similar. Note that Open Inventor does a limited form of automatic garbage collection even in C++. Therefore you can only create node objects using “new” and you are not allowed to explicitly delete node objects. If you are programming in C++ you should read the discussion of “reference counting” in the Inventor Mentor, but we won’t need to do anything specific about that in today’s simple example.



Viewer classes

- Integrate Open Inventor with window system / UI toolkit
- Derived from native window/widget classes
- Not required
- But… most applications use them to get:
  - Automatic setup/initialization of OpenGL window
  - Built-in mouse interface for manipulating camera
  - Built-in support for stereo, anti-aliasing, etc.



12

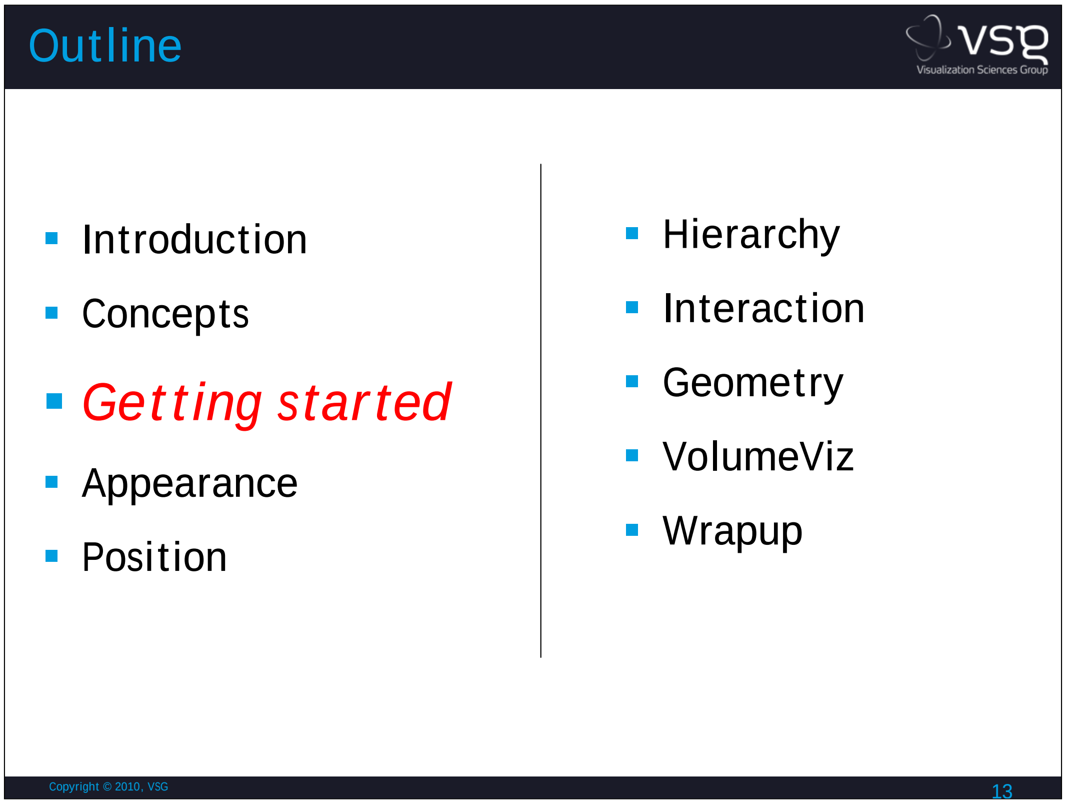
One of the powerful features of Open Inventor is the **viewer** classes. These classes integrate the 3D drawing window into the window system or user interface framework, making it easy to put one or more 3D windows where they belong in your user interface. There are specific versions of these classes for X11/Motif, Windows native, Qt, Wx, etc.

<u><CLICK!></u> In addition to this integration the viewer classes provide a lot of functionality that direct OpenGL programmers must spend time implementing or spend time searching for on the web. For example:

•Setting up and initializing an OpenGL window,

•Processing mouse events to provide navigation in the 3D scene, and

•Support for stereo, full screen rendering, antialiasing and more.



Outline

Let’s get started writing an Open Inventor program!

Application outline

1. Include C++ headers
   (import Java packages or .NET assemblies)
2. Create top level window/UI
   Initialize Open Inventor (C++ only)
3. Create Open Inventor scene graph
4. Create/configure viewer
5. Display window and begin event loop
6. In response to user actions,
   Add/delete/modify objects in scene graph
7. Cleanup

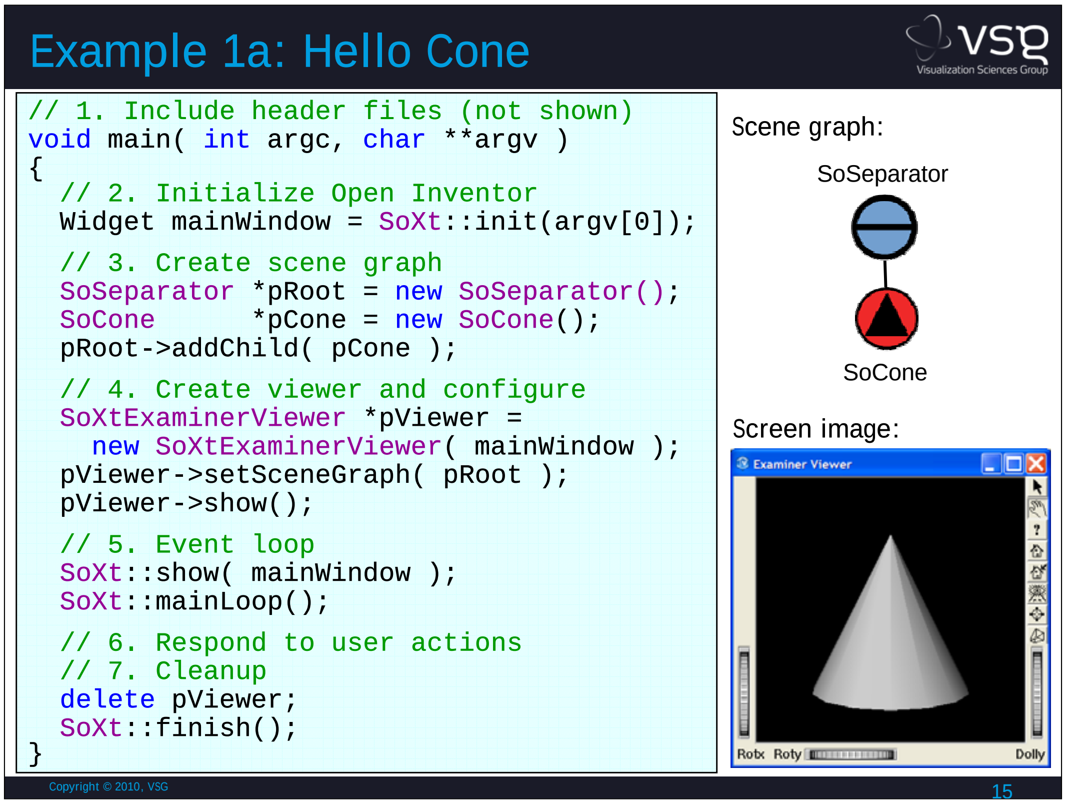
All Open Inventor applications follow the general outline shown here. The steps shown in red are specific to the Open Inventor API, so those are the things we'll focus on in this presentation.

First a few quick words about the steps we're **not** discussing. I assume that you have:

- Installed appropriate development tools, e.g. Visual Studio
- Downloaded and installed the appropriate Open Inventor SDK
- Obtained a license key or are on Windows and within the free trial period
- Created an appropriate project or make file (e.g. by copying one of the examples)

It's not that these steps are unimportant, but they are standard software development and specific to your environment. Our time is limited, so we're going to focus on design and implementation of the actual application.

Also a few words about languages and application frameworks. There are differences between C++ and C#, or even between MFC and Qt, as far as the application structure and how you create the user interface. So… we won't spend much time talking about that. With a few small exceptions we're going to talk about the Open Inventor classes you need and how you will use those classes in your application. Whichever language and UI system you use, you'll see that the OIV classes are the same and their usage is almost exactly the same.



Example 1a: Hello Cone

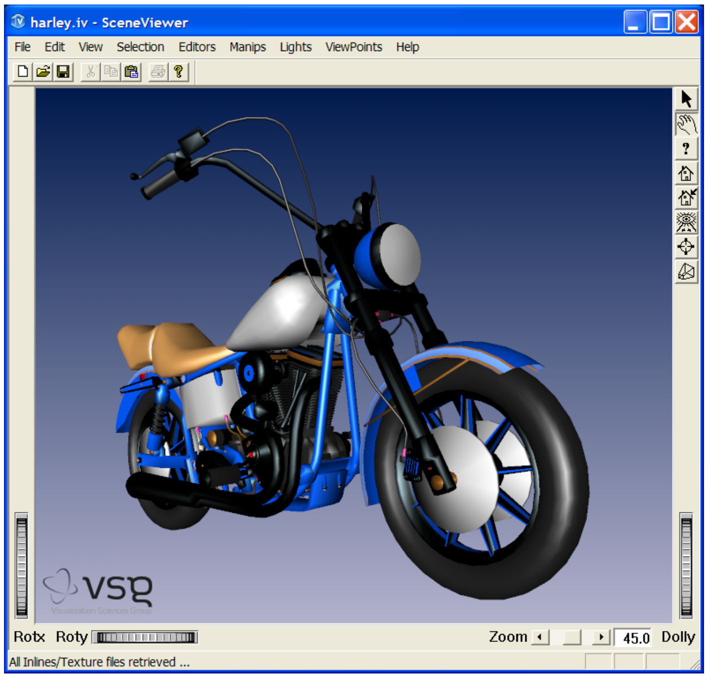
```
// 1. Include header files (not shown)
void main( int argc, char **argv )
{
  // 2. Initialize Open Inventor
  Widget mainWindow = SoXt::init(argv[0]);
  // 3. Create scene graph
  SoSeparator *pRoot = new SoSeparator();
  SoCone      *pCone = new SoCone();
  pRoot->addChild( pCone );
  // 4. Create viewer and configure
  SoXtExaminerViewer *pViewer =
    new SoXtExaminerViewer( mainWindow );
  pViewer->setSceneGraph( pRoot );
  pViewer->show();
  // 5. Event loop
  SoXt::show( mainWindow );
  SoXt::mainLoop();
  // 6. Respond to user actions
  // 7. Cleanup
  delete pViewer;
  SoXt::finish();
}
```

15

Here is a complete "Hello Cone" program in C++, a diagram of the scene graph it creates, and what the application looks like on the screen. Yes, a working, although trivial, Open Inventor application is this small. Not bad considering it typically takes several pages of code just to initialize the window when coding directly at the OpenGL level.

1. Step 1: Include header files for the Open Inventor classes we're going to use.
2. In step 2 we'll use the SoXt::init static method to do several things including initialize the Open Inventor library and create a top-level widget (or window) for the application. Here we used the **SoXt** set of classes, which are nominally for the X11/Xt/Motif user interface framework. However for convenience, Open Inventor automatically remaps these names to the corresponding **SoWin** names if you are building the program on Windows. This allows the same example code to work everywhere.
3. In step 3 we create the scene graph. In this minimal scene graph we'll only have a grouping node, called **SoSeparator**, and a shape node called SoCone. The hierarchical structure of the scene graph is defined by the addChild() method. Now when the Separator is traversed by an action, the Cone will also be traversed. You may have noticed that this simple scene graph does not contain a camera node or a light node, although they were shown in the very first scene graph diagram we looked at. That's because the viewer class will automatically create a default camera and a default light, called the *headlight*, for us.
4. In step 4 we create an instance of the most commonly used viewer class, the **ExaminerViewer**. There are other types of viewer, with different models for interpreting mouse events. The Examiner viewer is well suited for visualizing objects or data sets. It allows the user to move and rotate around the center of the scene (or any point you specify). We create the view as a "child", in the

Demo


harley.iv - SceneViewer
File Edit View Selection Editors Manips Lights ViewPoints Help
Rotx Roty
Zoom 45.0 Dolly

All Inlines/Texture files retrieved ...

Let's fire it up and look at some of the features that come "for free" with the Open Inventor viewer class. First I have to remind you that running over WebEx we have "interactive" performance, but not the same frame rate and image quality that you'll see running Open Inventor applications locally. I'm actually running a demo/example program called **SceneViewer** that comes with the Open Inventor SDK. First we'll display the "gray cone" scene graph that we created in the example. Now we'll load a more interesting model so the viewer features will be easier to see. This model also comes with the SDK, in the Open Inventor file format. [→ Recommend loading harley.iv] Notice that:

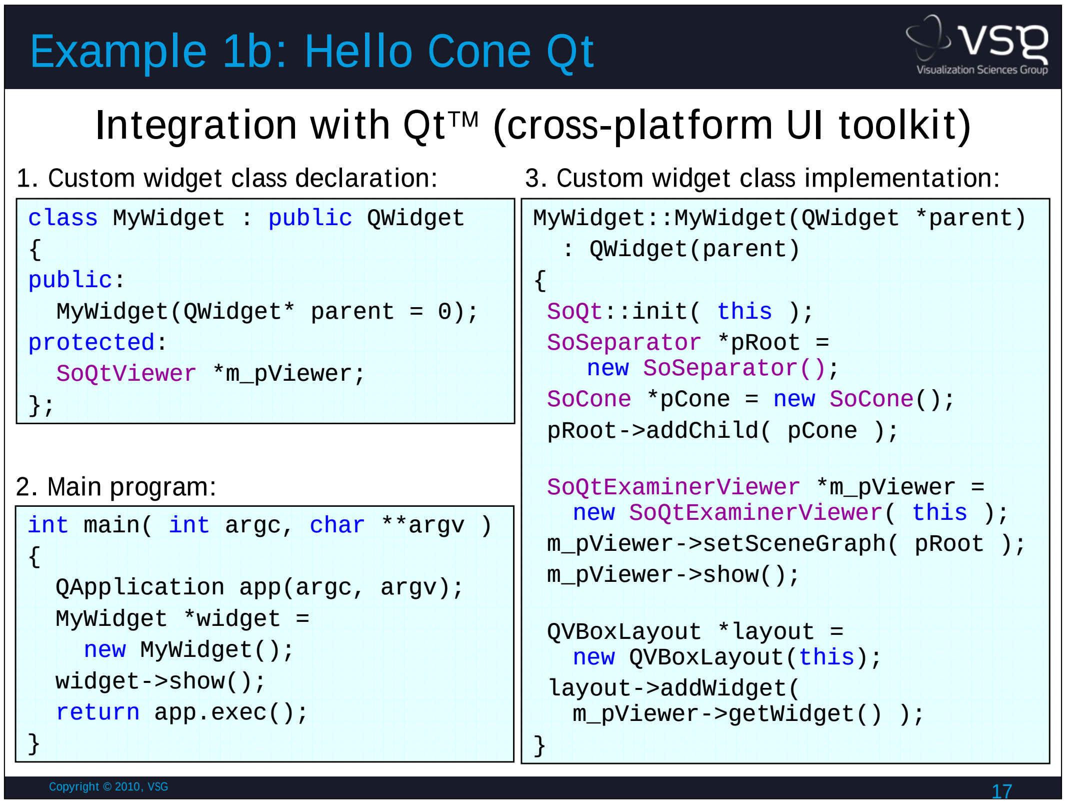
•The viewer allows us to **rotate, pan and dolly** the camera around the geometric center of the scene using the mouse and/or thumbwheels.

•The viewer also provides a **right-mouse menu** that includes all the button functions plus more options like full screen and stereo. Of course you can turn off the viewer's user interface in your actual application, but it's very convenient for prototyping and testing.

•The **Seek button** provides another powerful way to move the camera around. When the user clicks on an object the viewer moves in closer and makes that the center of rotation. [→ Recommend clicking on the gas tank.]

•In **selection mode** the user can select objects by clicking on them and Open Inventor will automatically highlight the selected objects. Here just using a red outline. [→ Recommend clicking on the gas tank.]

•Once an object is selected we can easily allow the user to modify the appearance of the object. Here using the default **Material Editor** provided with Open Inventor. For example we can make this object semi-transparent using the transparency slider. We'll see how to enable selection and modify object properties later in the presentation.



Example 1b: Hello Cone Qt

Integration with Qt™ (cross-platform UI toolkit)

1. Custom widget class declaration:

```
class MyWidget : public QWidget
{
public:
  MyWidget(QWidget* parent = 0);
protected:
  SoQtViewer *m_pViewer;
};
```

2. Main program:

```
int main( int argc, char **argv )
{
  QApplication app(argc, argv);
  MyWidget *widget =
    new MyWidget();
  widget->show();
  return app.exec();
}
```

3. Custom widget class implementation:

```
MyWidget::MyWidget(QWidget *parent)
  : QWidget(parent)
{
 SoQt::init( this );
 SoSeparator *pRoot =
    new SoSeparator();
 SoCone *pCone = new SoCone();
 pRoot->addChild( pCone );

 SoQtExaminerViewer *m_pViewer =
   new SoQtExaminerViewer( this );
 m_pViewer->setSceneGraph( pRoot );
 m_pViewer->show();

 QVBoxLayout *layout =
   new QVBoxLayout(this);
 layout->addWidget(
   m_pViewer->getWidget() );
}
```

Here is the "Hello Cone" program again. The platform independent version we previously looked at is not a novelty, it's actually very useful for test cases and many of the basic Open Inventor examples are written using this pattern. You can even add event handling or simple dialog boxes using platform independent features in Open Inventor (look for DialogViz if you're interested). But you'll probably use an application framework or a user interface framework, for your real application. In this example we'll see how easy it is to integrate Open Inventor into such a framework. One of the best, and most often used with Open Inventor, is **Qt** from Trolltech.

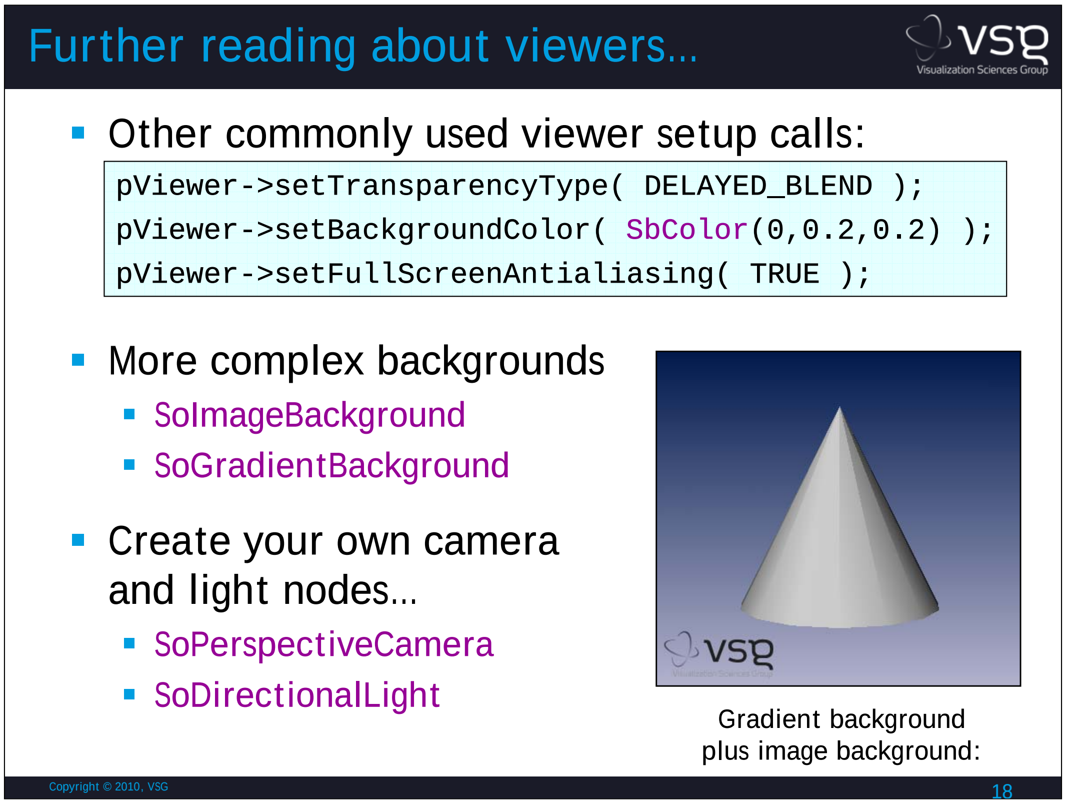
It looks like a lot more code, but you'll see that the Open Inventor part of the code is exactly the same. All the extra code is related to Qt. Let's start with a typical Qt application pattern, like the one in the standard Qt tutorial. The left side is pretty much the same as the Qt tutorial program.

-We derive our own widget class from QWidget.
This class must have a constructor, which we'll see in a minute, that creates the initial child widgets and so on. We just add a member variable that stores a pointer to our Open Inventor viewer object. For simplicity I've omitted some useful, but trivial to implement, things like an accessor function to get the viewer pointer.

-The main program is classic Qt and should be very familiar if you already use Qt. We initialize Qt by creating a QApplication object, create an instance of our custom widget (which will be the application's main window), make our widget visible, then start the event loop (which we did in the previous example using the SoXt static methods).

-Our custom widget's constructor looks remarkably like the code in the previous example. But in this case we use an alternate version of SoQt::init that assumes someone else is responsible for the application's main window. The scene graph creation code is exactly the same. The viewer creation and setup code is also the same except that we pass "this", our custom widget, as the parent widget. Finally there is a little bit of Qt to make the example work nicely. We create a "layout" object that, in this simple case, just automatically manages the size of the viewer widget to fill the main window. Notice that the SoQtViewer object is not itself a widget, but can give us its widget.

That's about all you need to know to start adding user interface elements and that's all we're going to say about windows, widgets and platform dependent stuff today. There is more information in the User's Guide and the many example programs. For the rest of the presentation we'll focus on using Open Inventor classes.

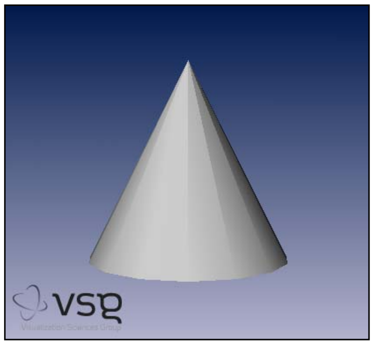


Further reading about viewers…

- Other commonly used viewer setup calls:

```
pViewer->setTransparencyType( DELAYED_BLEND );
pViewer->setBackgroundColor( SbColor(0,0.2,0.2) );
pViewer->setFullScreenAntialiasing( TRUE );
```

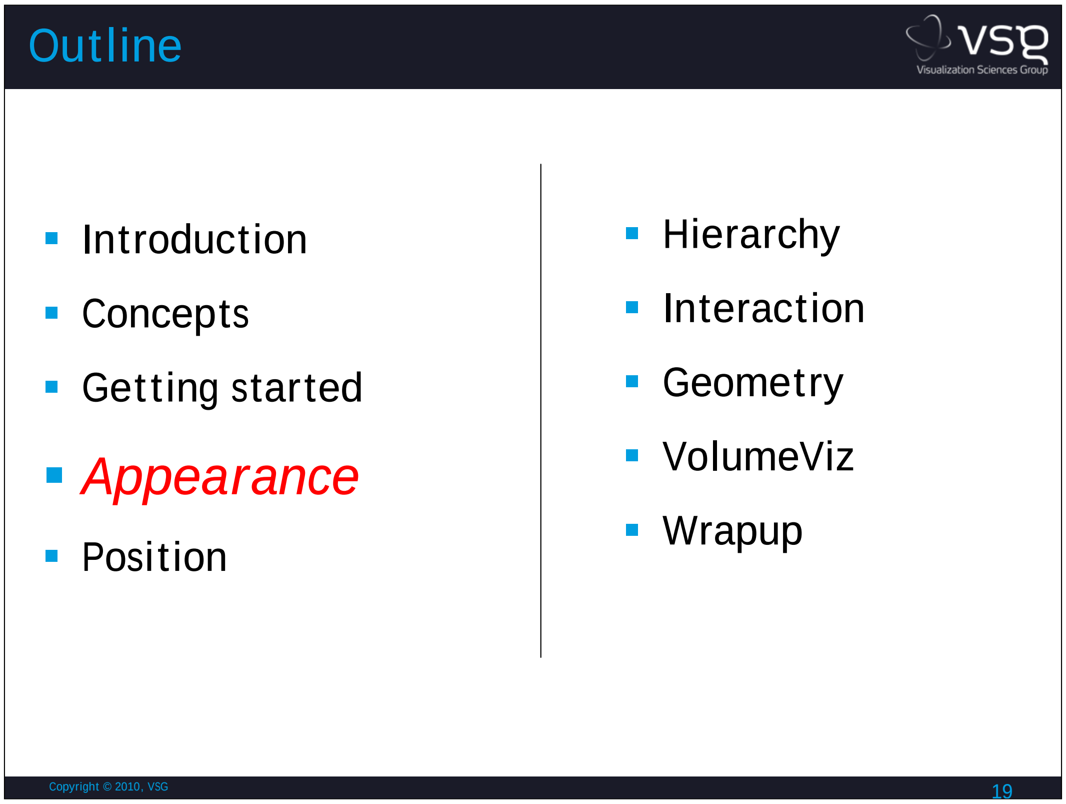
- More complex backgrounds
  - SoImageBackground
  - SoGradientBackground
- Create your own camera and light nodes…
  - SoPerspectiveCamera
  - SoDirectionalLight



Gradient background plus image background:

Our short example just “scratches the surface” of the features provided by the viewer classes. Here are just a few of the other things you may want to take advantage in your application. You can find more information about these features in the Open Inventor documentation. For example, the viewer object allows you to set its background color, to specify how transparent objects should be handled, and to enable full screen antialiasing to improve the image quality. We recommend at least setting the transparency type as part of your standard viewer setup.

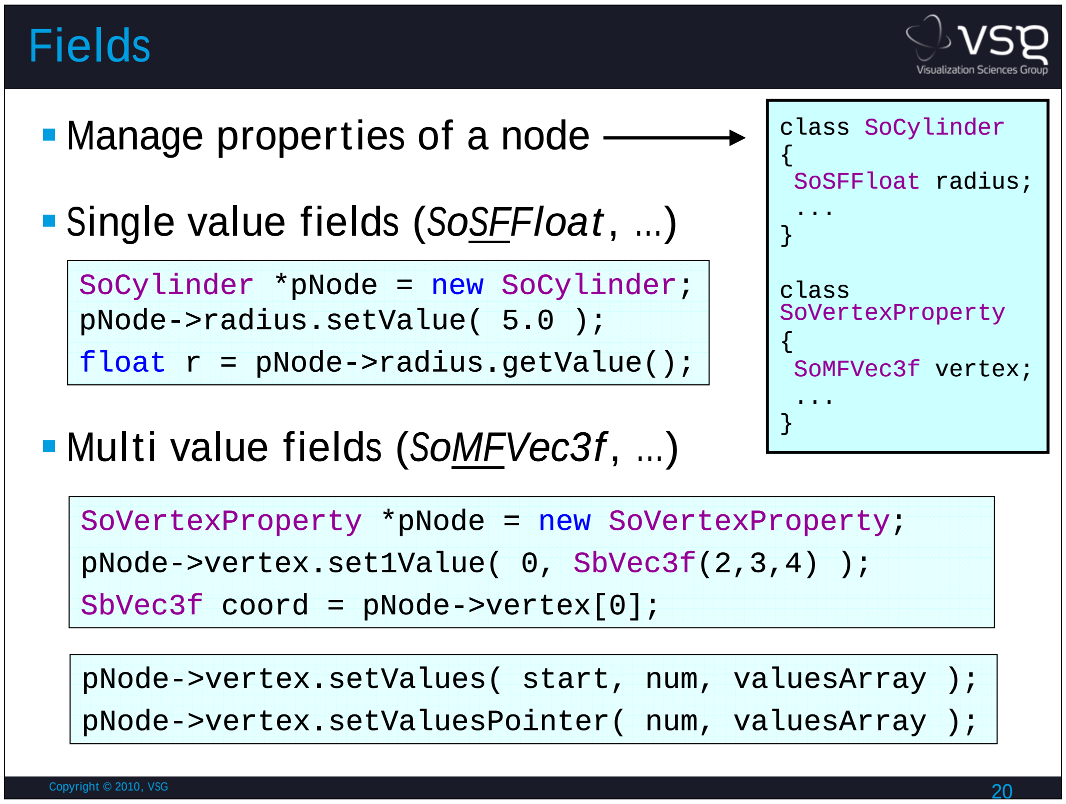
You can easily add more complex backgrounds using the SoBackground nodes. For example a color gradient can make the presentation of the scene much more interesting for your users and an image can be used as the background of the entire window or to place a logo in the corner of the window as we did here.



Outline

- Introduction
- Concepts
- Getting started
- *Appearance*
- Position
- Hierarchy
- Interaction
- Geometry
- VolumeViz
- Wrapup

We'll need a lot more properties than just gray for our actual objects. How do we specify the *appearance* of shapes in the scene?



Fields

- Manage properties of a node
- Single value fields (*SoSFFloat*, ...)

```
SoCylinder *pNode = new SoCylinder;
pNode->radius.setValue( 5.0 );
float r = pNode->radius.getValue();
```

- Multi value fields (*SoMFVec3f*, ...)

```
SoVertexProperty *pNode = new SoVertexProperty;
pNode->vertex.set1Value( 0, SbVec3f(2,3,4) );
SbVec3f coord = pNode->vertex[0];
```

```
pNode->vertex.setValues( start, num, valuesArray );
pNode->vertex.setValuesPointer( num, valuesArray );
```

```
class SoCylinder
{
 SoSFFloat radius;
 ...
}

class
SoVertexProperty
{
 SoMFVec3f vertex;
 ...
}
```

20

First we'll go a little deeper into the important concept of "fields". Fields are not just member variables. They are smart data containers with a lot of interesting and useful capabilities. Most importantly, like STL containers they automatically manage memory to contain whatever data you put into them. Some other features that we don't have time to talk about today include:

-Fields can write their contents to a stream, allowing scene graphs to be saved and restored,

-Fields can be connected to other fields to propagate changes through the scene graph, and

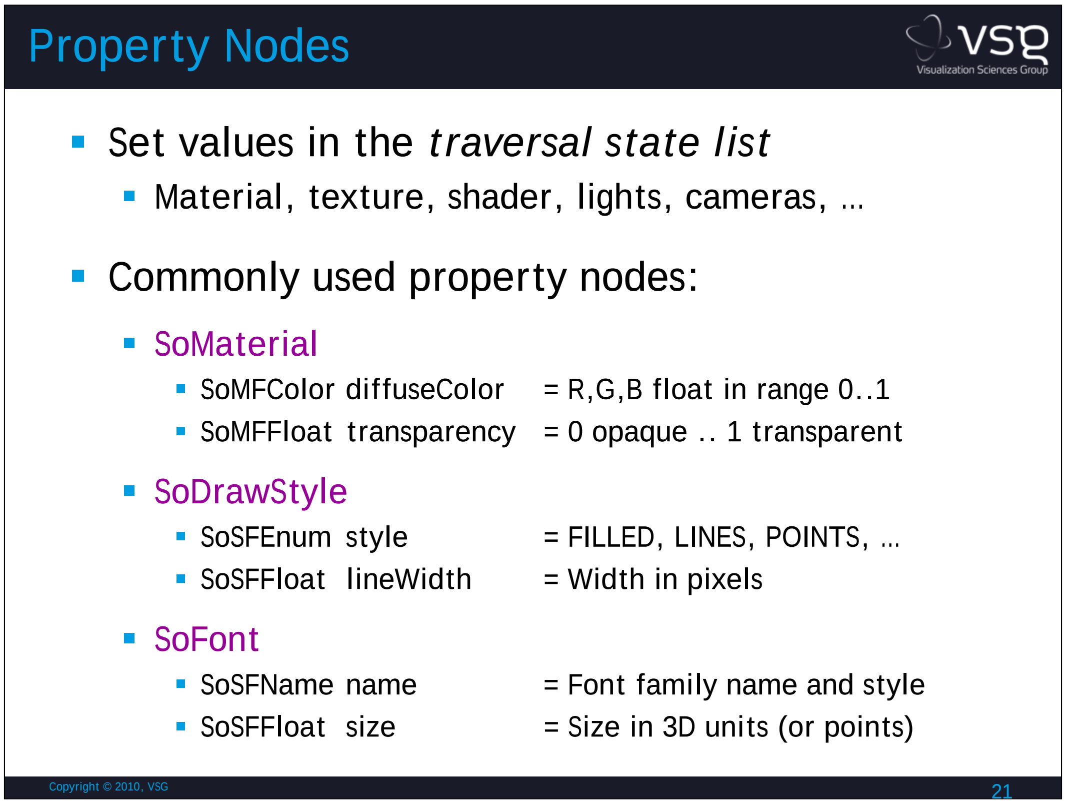
-Fields can notify when their contents are changed. See the **SoSensor** class for more details.

There are two general kinds of fields: single value fields with names beginning with SoSF and multiple value fields with names beginning with SoMF. Within these two groups there are separate classes for simple data types like int and float, common graphics types like points and vectors, as well as complex types like images. In the upper right you can see that the SoCylinder class, a predefined shape similar to SoCone, has a single valued float field named "radius" that specifies the radius of the cylinder.

In the first code fragment you can see the set and get methods. In C++ we also override the assignment operator for convenience, so you can also set the value using the "equals" sign.

<CLICK!> Next you can also see that the SoVertexProperty node has a multiple value field that contains "Vec3f" objects, in this case the coordinates of the vertices that define some shape. We have slightly different methods for multi-value fields. setValue still exists and just sets the first value in the field. set1Value takes the index of the value to set. We override the bracket operator to get the value at a specified index.

<CLICK!> We can set a range of values with one **setValues** call. This is very important for performance when setting a large number of values. Note that all these methods make a COPY of the application data. We can also use **setValuesPointer** to tell the field to use values that are stored in application memory, without making a copy.


Property Nodes

- Set values in the *traversal state list*
  - Material, texture, shader, lights, cameras, …
- Commonly used property nodes:
  - SoMaterial
    - SoMFColor diffuseColor = R,G,B float in range 0..1
    - SoMFFloat transparency = 0 opaque .. 1 transparent
  - SoDrawStyle
    - SoSFEnum style = FILLED, LINES, POINTS, …
    - SoSFFloat lineWidth = Width in pixels
  - SoFont
    - SoSFName name = Font family name and style
    - SoSFFloat size = Size in 3D units (or points)

21

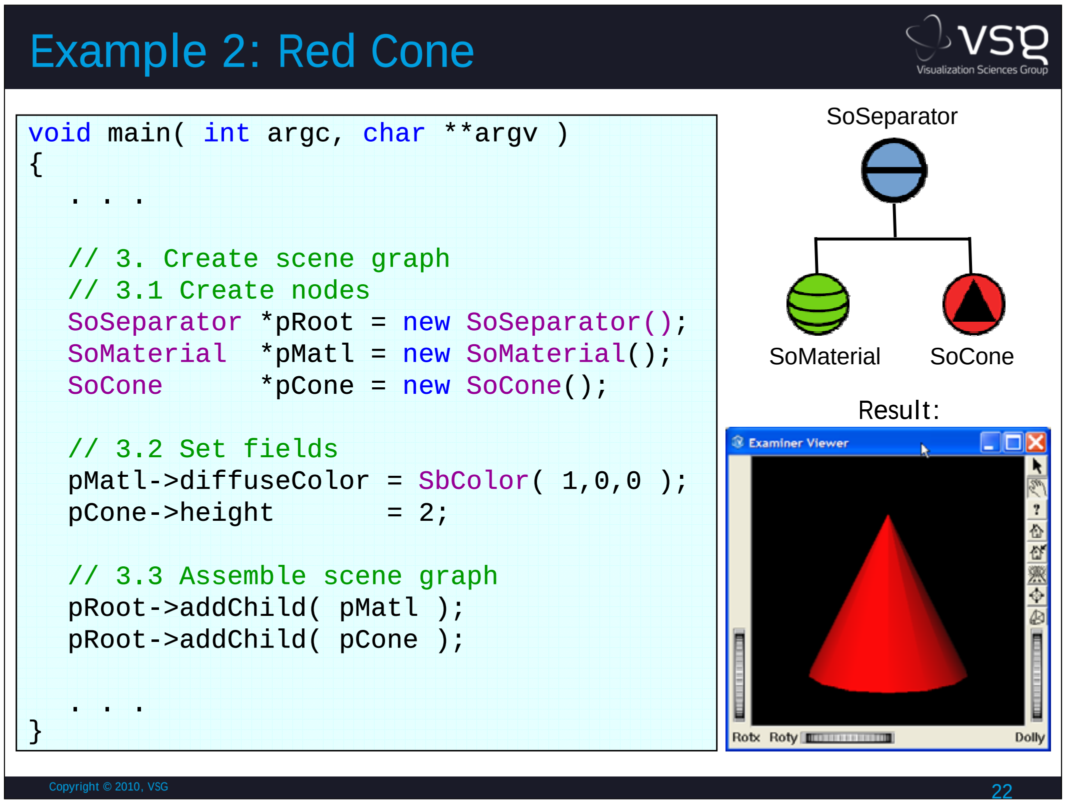
Armed with an understanding of fields, we can use many different kinds of property nodes to specify the appearance and behavior of nodes in the scene graph. We'll just mention a few of them here.

<u><CLICK!></u>

•SoMaterial contains a number of fields corresponding to the OpenGL material model. The most commonly used fields are *diffuseColor* and *transparency*. diffuseColor is specified with Red, Green and Blue float values ranging from 0 to 1. This is important because we may want to render with more than 8 bits per color component. Transparency is specified with a float value from 0 to 1 which is essentially the inverse of OpenGL's alpha component. In other words, 0 transparency means fully opaque and 1 means completely transparent. Note that in both cases the field can contain multiple values so we can apply different colors to different parts or vertices of a shape.

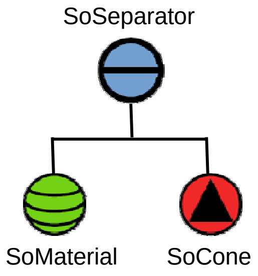
•SoDrawStyle sets a number of different things, for example the 'style' field controls how polygons are rendered, corresponding to OpenGL's polygonMode, the 'lineWidth' field controls how lines are rendered and so on.

•Another example is the SoFont node that specifies the font and size used to render text.

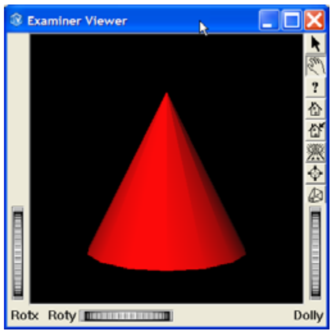


Example 2: Red Cone

```
void main( int argc, char **argv )
{
   . . .

   // 3. Create scene graph
   // 3.1 Create nodes
   SoSeparator *pRoot = new SoSeparator();
   SoMaterial  *pMatl = new SoMaterial();
   SoCone      *pCone = new SoCone();

   // 3.2 Set fields
   pMatl->diffuseColor = SbColor( 1,0,0 );
   pCone->height       = 2;

   // 3.3 Assemble scene graph
   pRoot->addChild( pMatl );
   pRoot->addChild( pCone );

   . . .
}
```

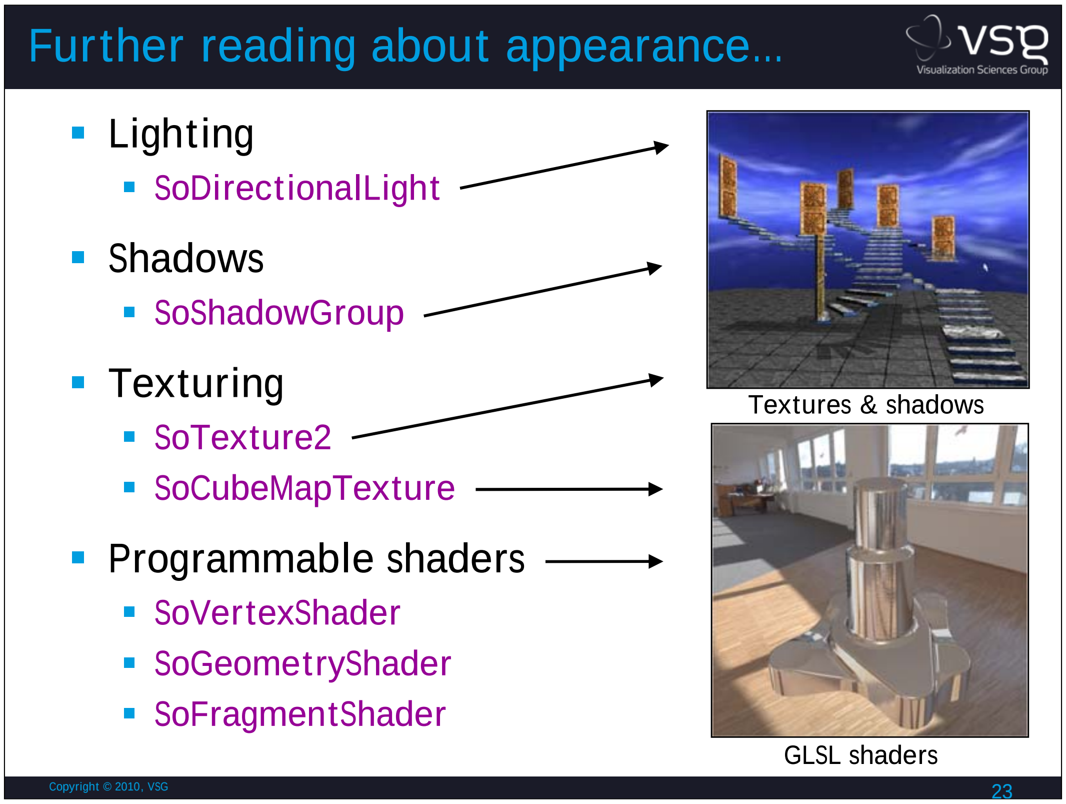


Result:



This is “Hello Cone” revisited, but I’ve omitted some parts of the program that are exactly the same as the previous example. We’re going to create one more node, the **SoMaterial** property node that we just discussed. We’ll set the diffuseColor field to full Red (no green or blue). We also set the height field of the cone just for illustration, although the value 2 is actually the default value. You can see the default values for all the fields of any node on its help page in the File Format section. Notice that we put the property node before the shape node because properties only affect nodes “downstream”. We’ll talk more about that in the next section of the tutorial. The resulting scene graph diagram and screen image are shown on the right.

Note that it doesn’t matter what order we create and initialize the nodes. The order above is just one possibility. However the order of the addChild calls determines the sequence of the nodes in the scene graph and that is important as you’ll see in the slides ahead.



Further reading about appearance…

- Lighting
  - SoDirectionalLight
- Shadows
  - SoShadowGroup
- Texturing
  - SoTexture2
  - SoCubeMapTexture
- Programmable shaders
  - SoVertexShader
  - SoGeometryShader
  - SoFragmentShader

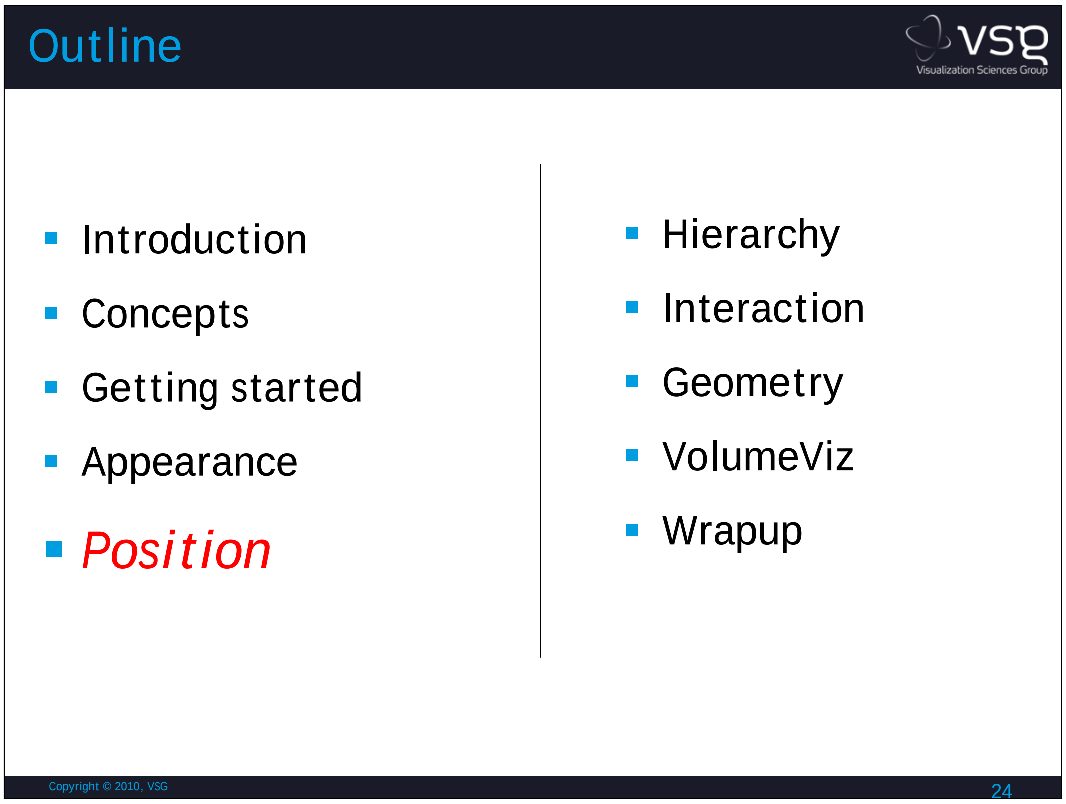
Textures & shadows

GLSL shaders

Copyright © 2010, VSG

23

This short example just "scratches the surface" of the features provided by the property classes. Here are just a few of the other features you may want to use in your application. You can find more information about these features in the Open Inventor documentation.

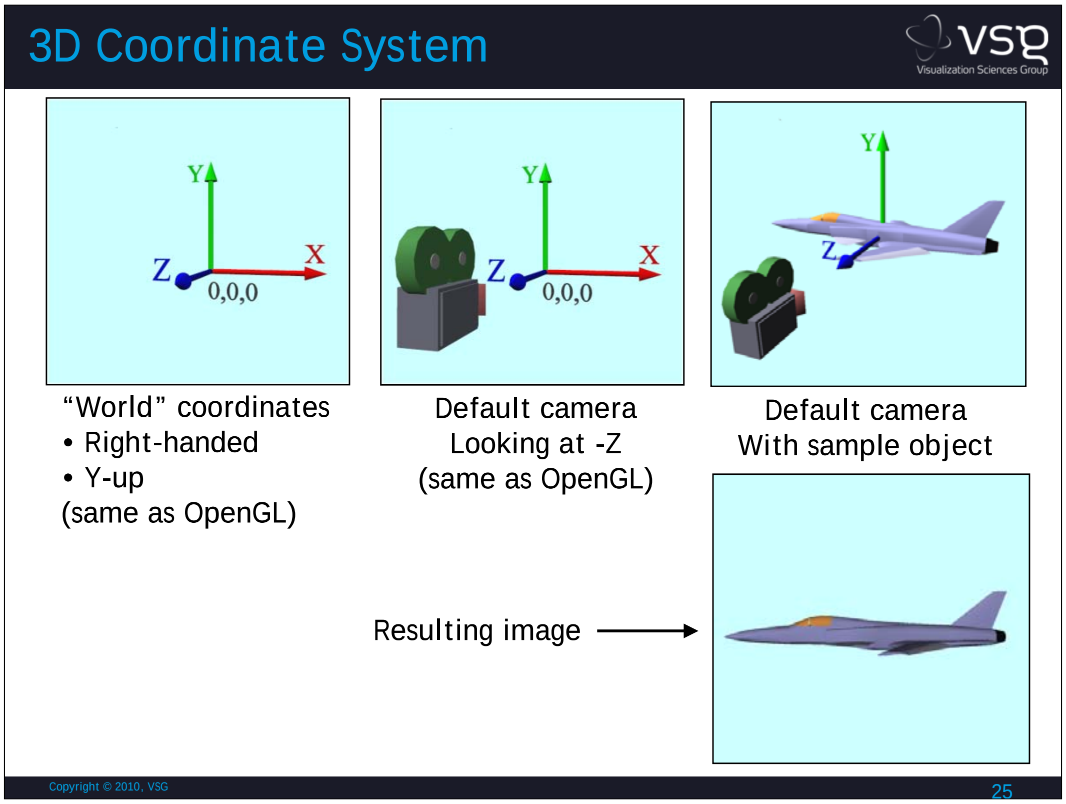


Outline

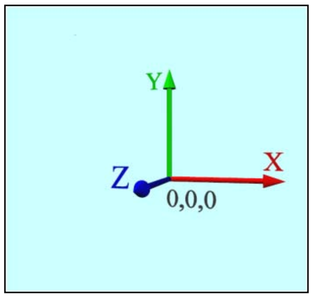
- Introduction
- Concepts
- Getting started
- Appearance
- *Position*
- Hierarchy
- Interaction
- Geometry
- VolumeViz
- Wrapup

24

We’ll probably need more than one object in the scene graph. How do we position objects in 3D space?

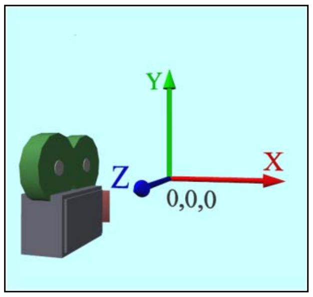


3D Coordinate System

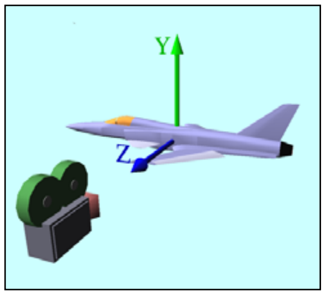


"World" coordinates
- Right-handed
- Y-up

(same as OpenGL)



Default camera
Looking at -Z
(same as OpenGL)



Default camera
With sample object

Resulting image →

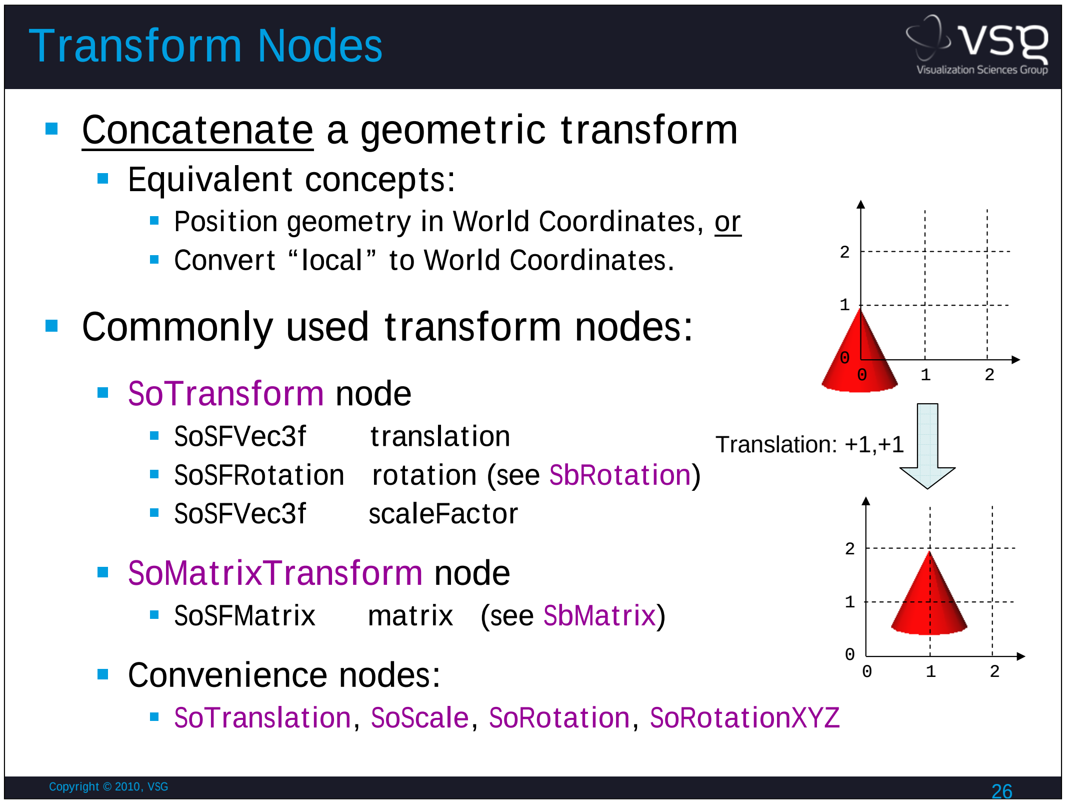
25

First we'll need a clear understanding of how 3D space is defined. Open Inventor uses the same coordinate system as OpenGL, a so-called "right handed" coordinate system, shown here.

<u><CLICK!></u> The default properties of the camera node also define the same view of 3D space as OpenGL. +X is to the right, +Y is "up" and +Z points toward the viewer. We call this the "world" coordinate system, as opposed to the "local" coordinate system relative to a specific geometry. The second image shows the default position and orientation of the camera.

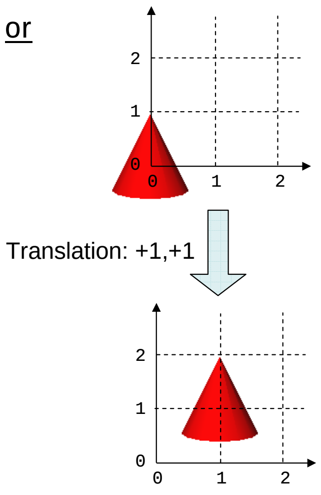
<u><CLICK!></u> The third image shows a "birds eye" view of the 3D space with a sample object and the default camera and the bottom image shows what the camera "sees".

<u><CLICK!></u> The default camera is set to view the range -1 to 1 in X, Y and Z, but your scene probably has a different extent in 3D. You may be wondering how to initially position the camera so that the entire scene is visible. In many cases it's automatic or at least very easy. The viewer provides a *viewAll* method that automatically moves the camera so the entire scene is visible. In fact if you allow the viewer to automatically create the camera, it will also automatically call *viewAll* in the *setSceneGraph* method. This is how the simple example works.



Transform Nodes

- Concatenate a geometric transform
  - Equivalent concepts:
    - Position geometry in World Coordinates, or
    - Convert "local" to World Coordinates.
- Commonly used transform nodes:
  - SoTransform node
    - SoSFVec3f translation
    - SoSFRotation rotation (see SbRotation)
    - SoSFVec3f scaleFactor
  - SoMatrixTransform node
    - SoSFMatrix matrix (see SbMatrix)
  - Convenience nodes:
    - SoTranslation, SoScale, SoRotation, SoRotationXYZ

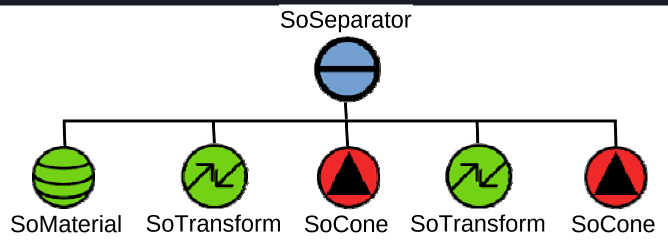


26

The transform nodes *concatenate* a geometric transform onto the current modeling matrix. By this we mean that multiple transform nodes "accumulate" their effect.

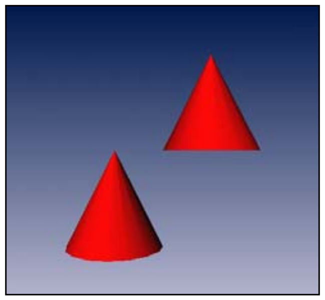
<u><CLICK!></u> You can think of the current modeling matrix as either positioning geometry in World Coordinates or as converting the object's "local" coordinate system to World Coordinates. These are two equivalent, but inverse views of the same thing. This is the same as the modelView matrix in OpenGL except that, for convenience, Open Inventor maintains separate model and view matrices. So, for example, the actual coordinates defining the geometry of an SoCone are centered around the point 0,0,0, but if we apply a *translation* of +1,+1 to the default red cone in the upper image, the result is that the cone is positioned at 1,1.
<u><CLICK!></u> **SoTransform** allows you to specify scale, rotation and translation values separately. You can also use convenience nodes like **SoTranslation** which only contain a translation field. There is no performance difference, but the resulting code may be slightly more maintainable because the semantics imply that only translation is valid for that object. You can also use **SoMatrixTransform** to apply an arbitrary 4x4 matrix. If you are not already familiar with the concepts of geometric transformations, I recommend reading more about them in the Inventor Mentor.

Example 3: More Cones



```
#include <Inventor/nodes/SoTransform.h>
. . .
   SoTransform *pTrn1 = new SoTransform();
   SoTransform *pTrn2 = new SoTransform();
   pTrn1->translation = SbVec3f(-1,-1,0 );
   pTrn2->translation = SbVec3f( 2, 2,0 );

   pRoot->addChild( pMatl );
   pRoot->addChild( pTrn1 );
   pRoot->addChild( pCone );
   pRoot->addChild( pTrn2 );
   pRoot->addChild( pCone );  //Reuse node
. . .
```

Result:



27

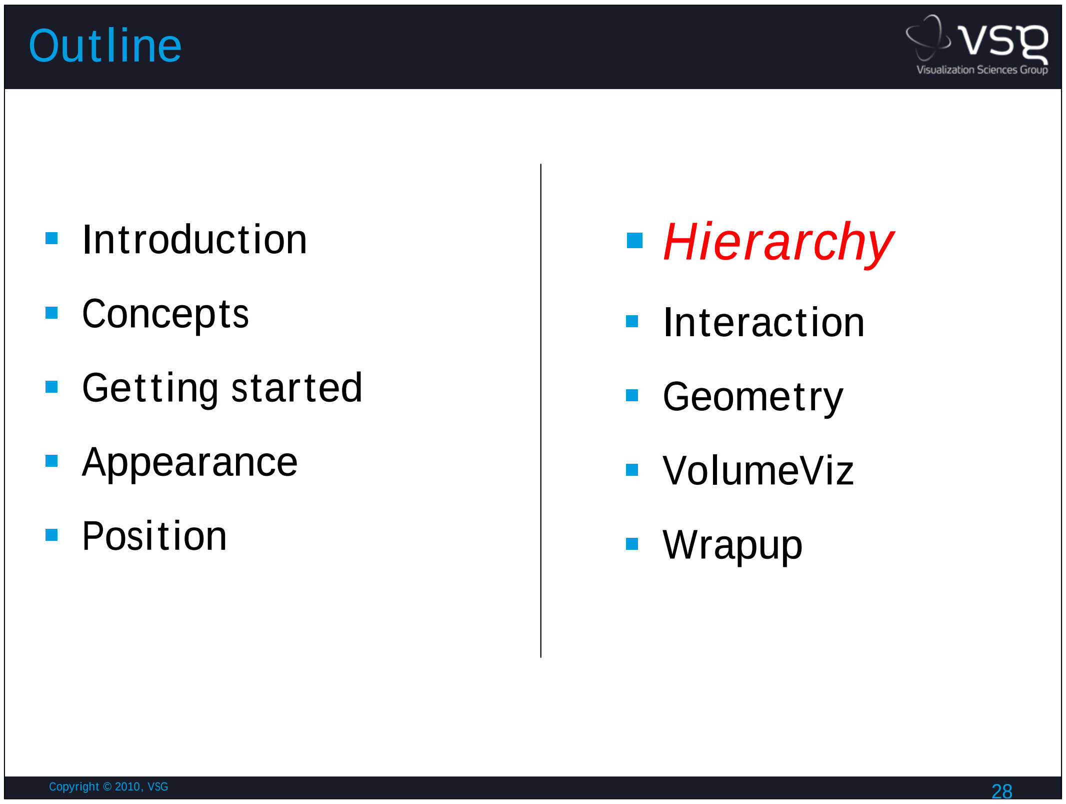
Let's make a scene with multiple cones. If we just created two SoCone nodes and added them to the scene graph, the resulting image would be exactly the same as HelloCone, because the cones would be positioned at exactly the same place!

Instead we'll create an SoTransform to position each cone. Let's say we want to position the first cone at -1,-1,0 and the second cone at 1,1,0.

pTrn1 is traversed first, so the first cone will be positioned at -1,-1,0. But remember that transforms *concatenate*. So if we just set pTrn2 to 1,1,0 the second cone would be positioned at 0,0,0, which is not the desired result. We have to set pTrn2 to 2,2,0 so that the sum of the translations is the desired 1,1,0. Having an object's position (or rotation) be relative to the preceding object can be useful in some cases, but it would be tedious for a large number of objects if we were just trying to position them. We'll see in the next section how to solve this problem.

Before we leave, notice two other things about this example. As long as both cones should be the same color, we only need one SoMaterial node. Both cones inherit the color Red in this case. If we wanted each cone to have a different color we would need more Material nodes and we'll see in the next section how to group them together in a logical fashion.

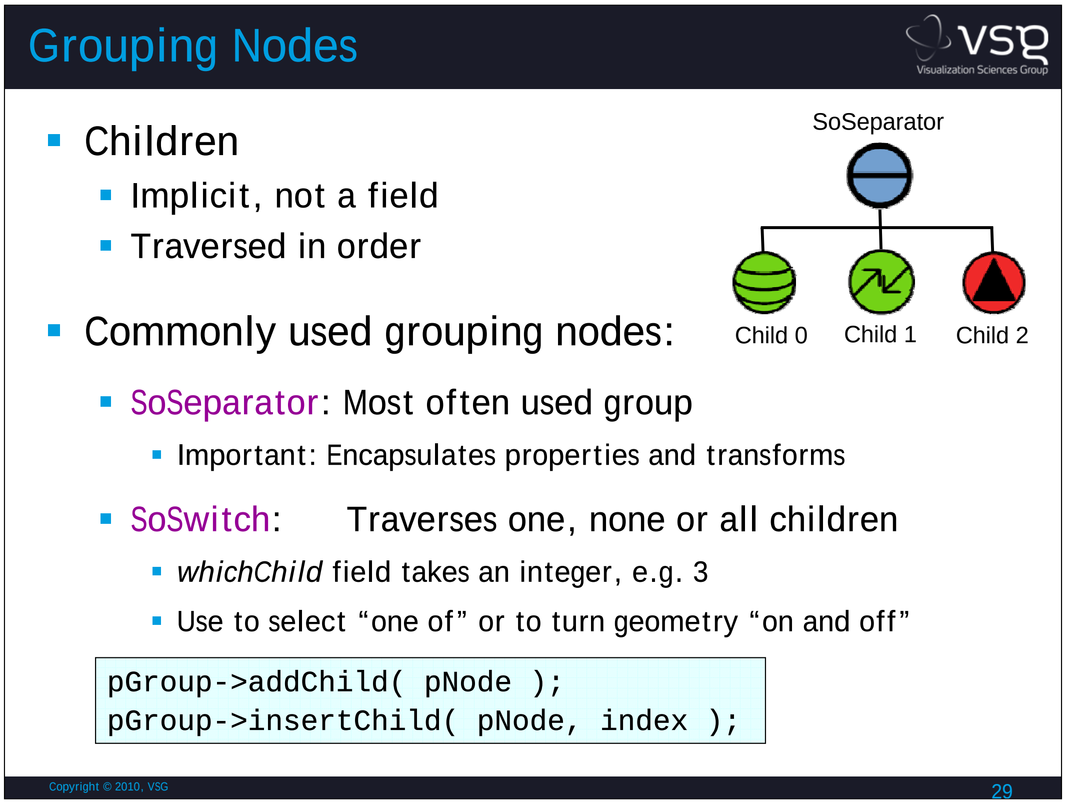
Second, notice that we only created one Cone object (that line of code is the same as the previous example and not repeated here) and added it to the scene graph multiple times. We call this "instancing" and each occurrence of the cone in the scene graph is an "instance" of the cone object. If we have a large number of objects in the scene that actually have the same geometry, instancing is a valuable technique for reducing the total memory footprint of the scene graph. For example if we need 1000 spheres in the scene, we don't need to create 1000 sphere objects, just have 1000 instances.



Outline

- Introduction
- Concepts
- Getting started
- Appearance
- Position
- *Hierarchy*
- Interaction
- Geometry
- VolumeViz
- Wrapup

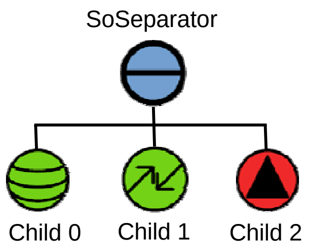
We can already see that it would make sense to group an object and its properties together. How can we do that in the scene graph?



Grouping Nodes

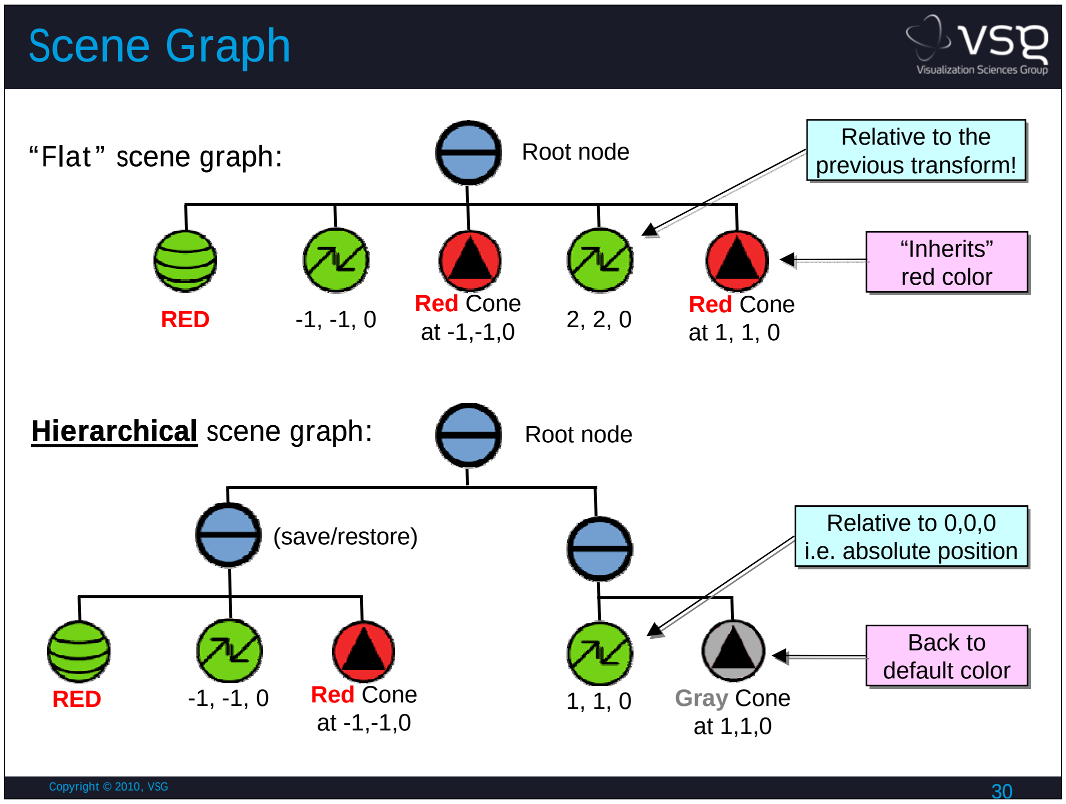
- Children
  - Implicit, not a field
  - Traversed in order
- Commonly used grouping nodes:
  - SoSeparator: Most often used group
    - Important: Encapsulates properties and transforms
  - SoSwitch: Traverses one, none or all children
    - *whichChild* field takes an integer, e.g. 3
    - Use to select "one of" or to turn geometry "on and off"

```
pGroup->addChild( pNode );
pGroup->insertChild( pNode, index );
```



The answer is to use grouping nodes. Every grouping node is derived from **SoGroup** and has zero or more children, which are traversed in a fixed order. The diagram in the upper corner shows a small scene graph that we could think of as a logical "entity" with its own geometry and its own material (e.g. color). ***<u><CLICK!></u>*** The most commonly used grouping node is **SoSeparator**. In addition to maintaining a list of children, this node "separates" state changes below it in the scene graph, or you could say keeps those state changes from "leaking out" into the rest of the scene graph. This is very useful because it allows us to specify the properties of an entity independent of other entities. Of course it could still inherit other properties that are not explicitly specified for it.

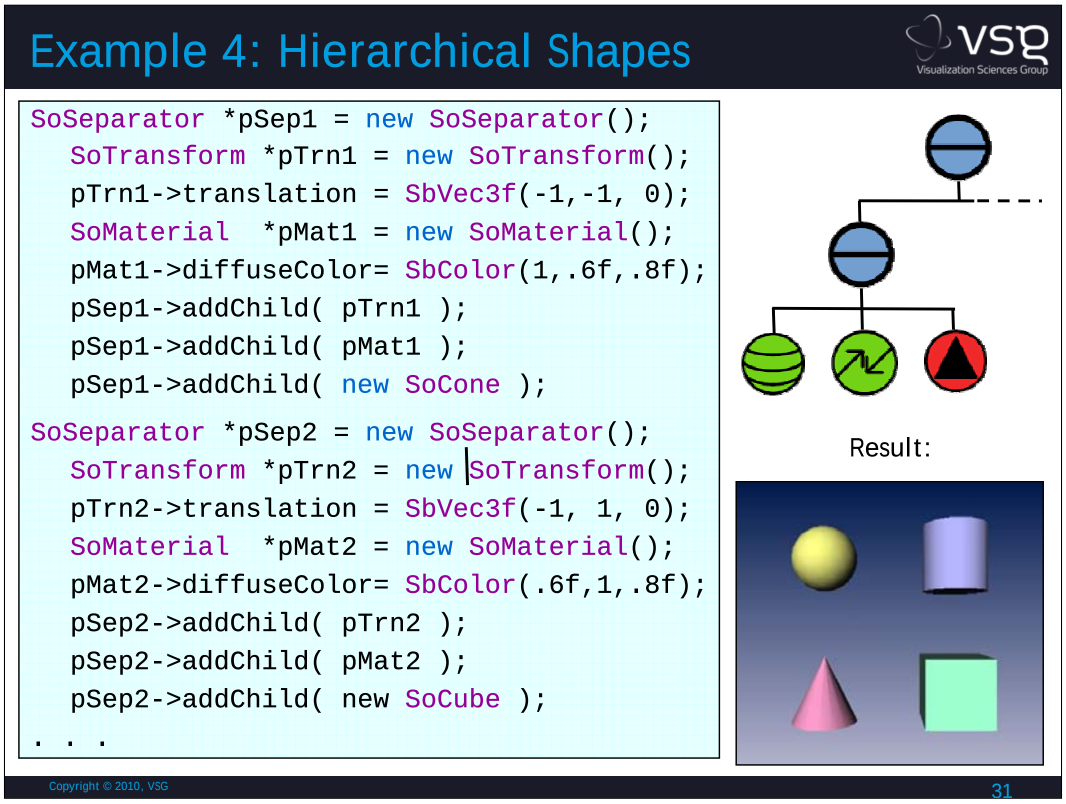
Another useful grouping node is the **SoSwitch**. SoSwitch can have multiple children like any grouping node, but its *whichChild* field specifies whether to traverse zero, exactly one, or all of the children. The special value -1 means to traverse none of the children and the special value -3 means to traverse all the children. This is very useful to toggle the visibility of an entity. You could accomplish the same visual effect by removing the entity from the scene graph and adding it back in, but this is inefficient and all rendering optimization information is lost if the entity is removed from the scene graph. The ability to specify one specific child to be traversed can be used, for example, to implement different versions of an entity or to implement an animation.



Scene Graph

Here is a visual summary of the things we just discussed about the scene graph. . First a "flat" scene graph. Notice that the second transform is relative to the first one and the second cone inherits the color red.

<u><CLICK!></u> Next a hierarchical scene graph. Here the properties of the first entity have been "separated" from the rest of the scene graph by an **SoSeparator** node. Pretty much all "real" scene graphs are hierarchical.



Example 4: Hierarchical Shapes

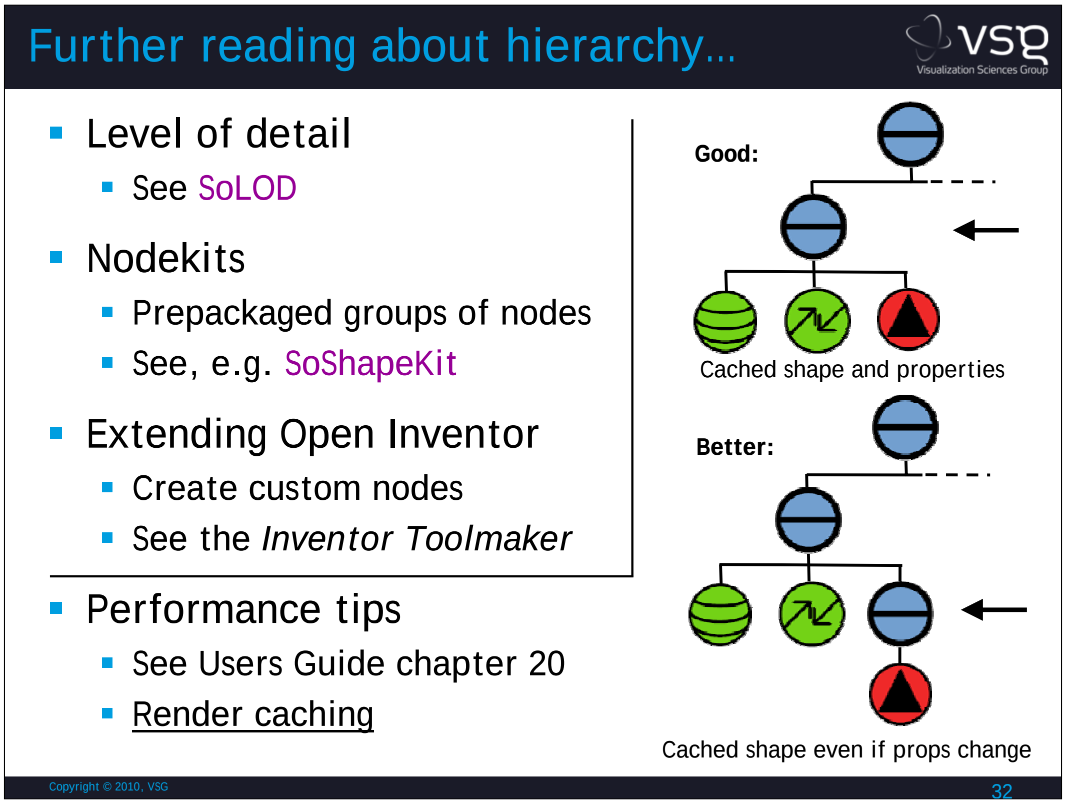
```
SoSeparator *pSep1 = new SoSeparator();
  SoTransform *pTrn1 = new SoTransform();
  pTrn1->translation = SbVec3f(-1,-1, 0);
  SoMaterial  *pMat1 = new SoMaterial();
  pMat1->diffuseColor= SbColor(1,.6f,.8f);
  pSep1->addChild( pTrn1 );
  pSep1->addChild( pMat1 );
  pSep1->addChild( new SoCone );
SoSeparator *pSep2 = new SoSeparator();
  SoTransform *pTrn2 = new SoTransform();
  pTrn2->translation = SbVec3f(-1, 1, 0);
  SoMaterial  *pMat2 = new SoMaterial();
  pMat2->diffuseColor= SbColor(.6f,1,.8f);
  pSep2->addChild( pTrn2 );
  pSep2->addChild( pMat2 );
  pSep2->addChild( new SoCube );
. . .
```

Result:

Let’s say we want to create multiple geometric entities in our scene graph and each entity can have its own position and color. What would the code to do that look like?

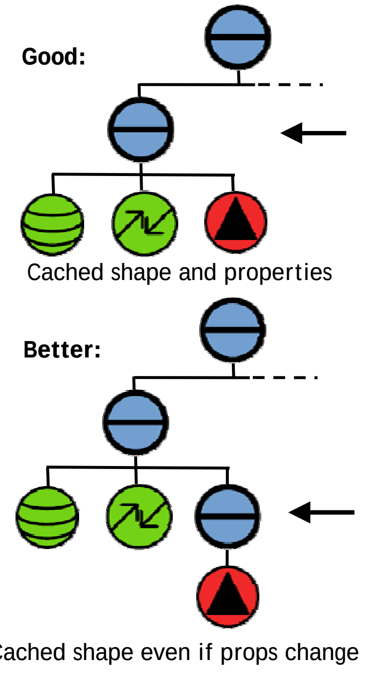
Each entity’s scene graph will look like the diagram in the upper right, consisting of a Separator node to encapsulate the properties, a Material node to specify the color, a Transform node to specify the position (rotation, scale factor, etc), and a shape node to render the actual geometry. Just for illustration we’ll create a sphere, a cylinder, a cone and a cube, understanding that these could be trees or airplanes or whatever we need to render.

The corresponding code should be quite easy to understand now.



Further reading about hierarchy…

- Level of detail
  - See SoLOD
- Nodekits
  - Prepackaged groups of nodes
  - See, e.g. SoShapeKit
- Extending Open Inventor
  - Create custom nodes
  - See the *Inventor Toolmaker*
- Performance tips
  - See Users Guide chapter 20
  - Render caching

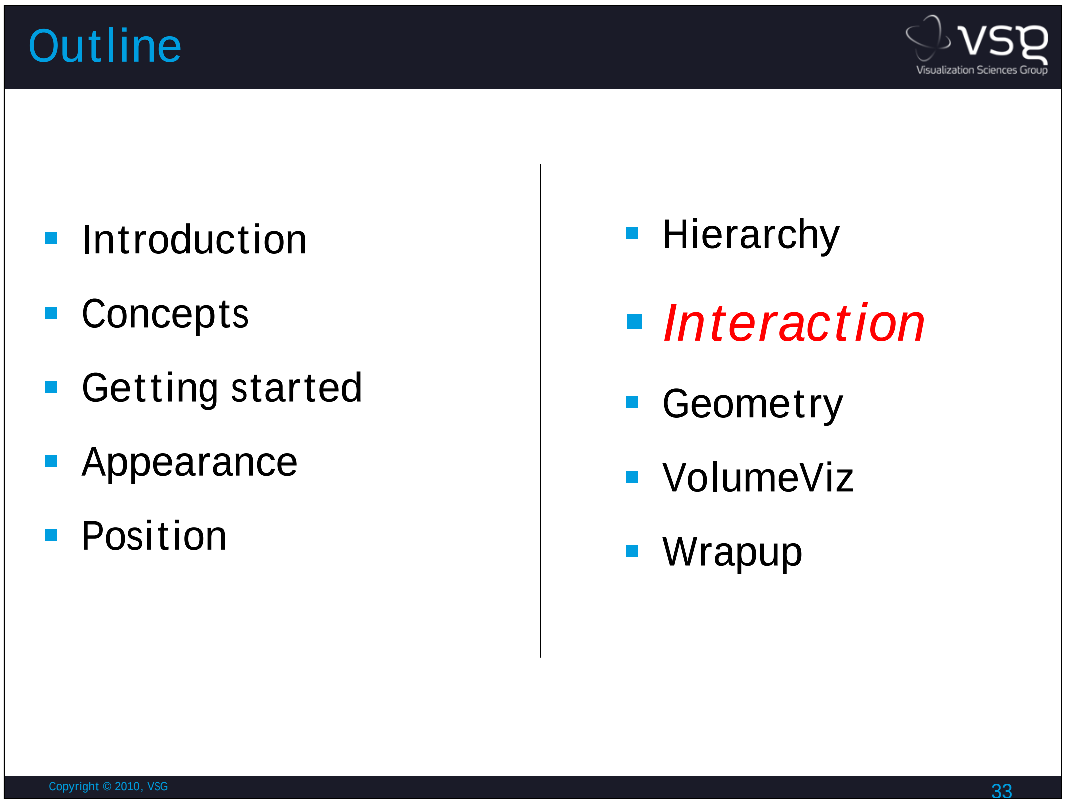

Good:
Cached shape and properties
Better:
Cached shape even if props change

32

This short section just “scratches the surface” of the features and issues involved in structuring the scene graph. Here are just a few of the other things you may want to take advantage in your application. You can find more information about these features in the Open Inventor documentation.

<CLICK!> I’ll just say a few words here about hierarchy and performance. In addition to encapsulating state changes, the Separator node can automatically build a “render cache” that represents its children in a highly optimized form that renders very quickly. Of course if the child nodes are modified then the cache must be re-built. The transform node, for example, might be modified quite often if we are using it to animate the shape.

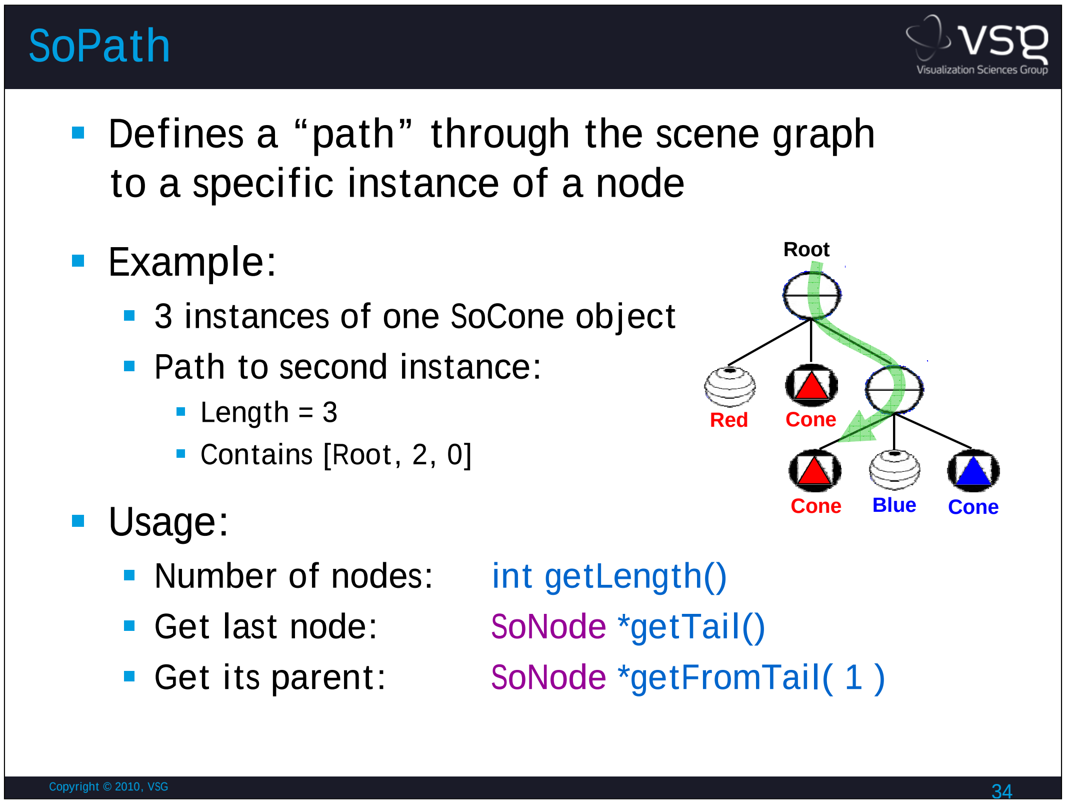
<CLICK!> In this case it’s better to add an additional Separator node that caches just the geometry, because this cache will be not be affected by changes to the property and transform nodes. There are other ways to optimize rendering, including vertex arrays and vertex buffer objects. You can read about these in the Open Inventor documentation.



Outline

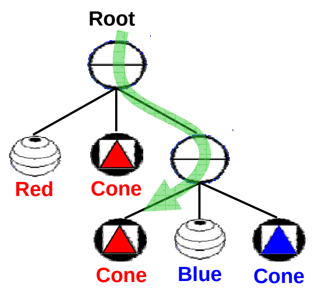
- Introduction
- Concepts
- Getting started
- Appearance
- Position
- Hierarchy
- *Interaction*
- Geometry
- VolumeViz
- Wrapup

Now that we have a scene graph and can view it on the screen, how do we allow the user to interact with the scene?



SoPath

- Defines a "path" through the scene graph to a specific instance of a node
- Example:
  - 3 instances of one SoCone object
  - Path to second instance:
    - Length = 3
    - Contains [Root, 2, 0]

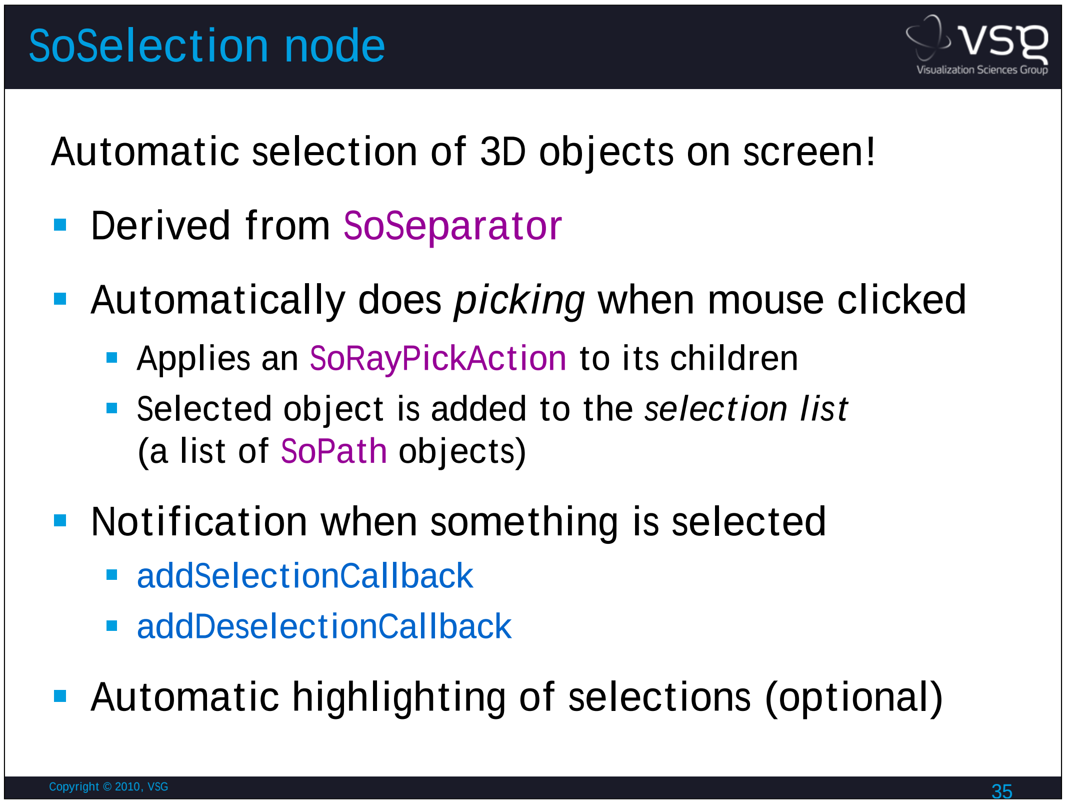


- Usage:
  - Number of nodes: int getLength()
  - Get last node: SoNode *getTail()
  - Get its parent: SoNode *getFromTail(1)

We've already discussed the difference between the node object itself and an instance of that object in the scene graph and that there may be multiple instances. An **SoPath** object is the way we identify one specific instance in the scene graph. We do that by defining a "path" from the top of the scene graph, usually called the "root" node, down through various grouping nodes to the specific instance.

<u><CLICK!></u> The diagram on the right shows the logical structure of our example scene graph. Logically there are 3 cones in the scene graph, but let's say that for efficiency we only created a single SoCone object and added it to the scene graph 3 times. A pointer, i.e. the address of the SoCone object, does not identify a specific instance, even though we have a clear mental concept of the "first red cone", the "second red cone" and the "blue cone". We can uniquely identify a specific instance, for example the "second red cone", using an SoPath. The contents of an SoPath are kind of like driving directions. It contains the starting point or "head" of the path, which is the root node in this case, followed by a list of child numbers, in this case meaning to traverse child 2, then traverse child 0, and so on until we reach the "tail" of the path.

<u><CLICK!></u> From a path object we can query many things, but particularly useful are the length of the path, the "tail", which is the last node in the path and usually the geometry that the user selected, and the parent of that node, which is 1 node above the tail and is always a group node.



SoSelection node

Automatic selection of 3D objects on screen!

- Derived from SoSeparator
- Automatically does *picking* when mouse clicked
  - Applies an SoRayPickAction to its children
  - Selected object is added to the *selection list* (a list of SoPath objects)
- Notification when something is selected
  - addSelectionCallback
  - addDeselectionCallback
- Automatic highlighting of selections (optional)

Copyright © 2010, VSG 35

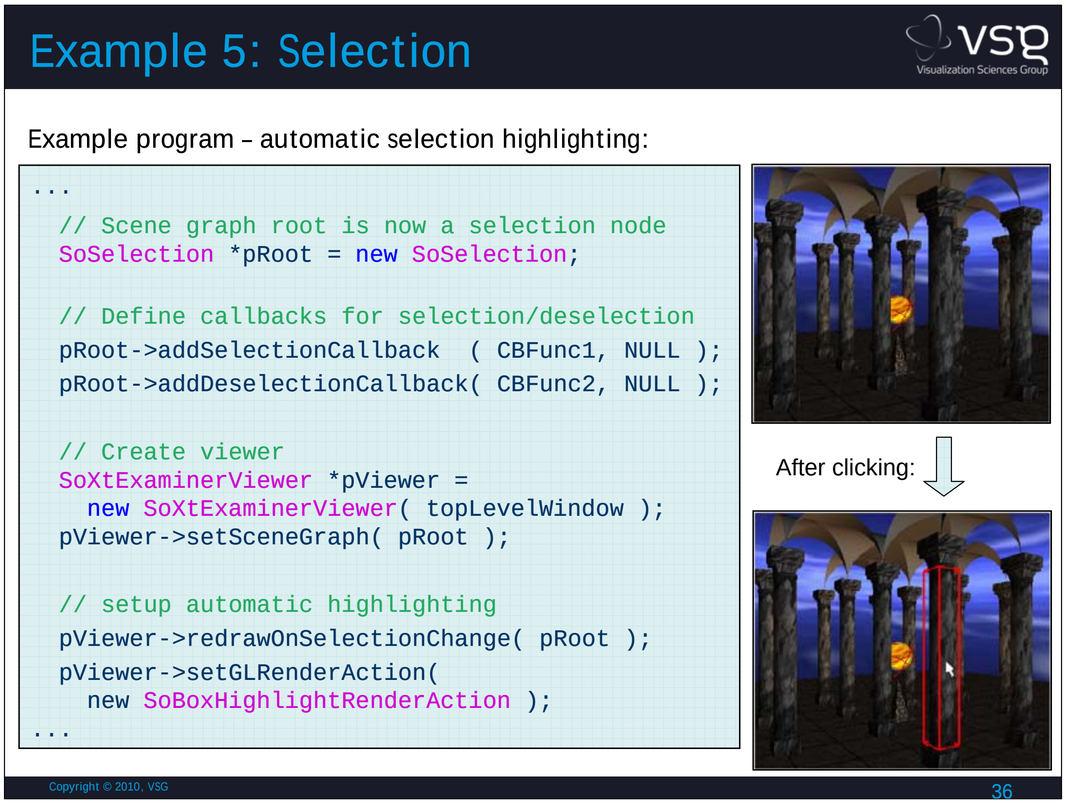
Let’s start at a high level, using an **SoSelection** node. This is a group node, derived from SoSeparator, that provides automatic selection of 3D objects on the screen.

<CLICK!> When the user clicks in the 3D window, the SoSelection node automatically does “picking”, sometimes called “hit testing”, meaning that it applies an **SoRayPickAction** to the scene graph which finds all the entities whose geometry is under the cursor and determines the closest one. For more information about picking and how to manage this directly, see the **SoRayPickAction** class and its associated example programs.

<CLICK!> If the pick action is successful, the result is an **SoPath** that identifies a specific instance and SoSelection puts this path in the *selection list*. Depending on the selection mode, this may add to the list or replace the previously selected object, causing the previous object to be de-selected.

When an object becomes selected or is de-selected, SoSelection notifies the application by calling one or more callback functions.

<CLICK!> There is even a mechanism that provides automatic highlighting of the selected entity or entities.

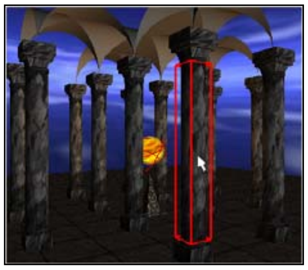


Example 5: Selection

Example program – automatic selection highlighting:

```
...
  // Scene graph root is now a selection node
  SoSelection *pRoot = new SoSelection;

  // Define callbacks for selection/deselection
  pRoot->addSelectionCallback  ( CBFunc1, NULL );
  pRoot->addDeselectionCallback( CBFunc2, NULL );

  // Create viewer
  SoXtExaminerViewer *pViewer =
    new SoXtExaminerViewer( topLevelWindow );
  pViewer->setSceneGraph( pRoot );

  // setup automatic highlighting
  pViewer->redrawOnSelectionChange( pRoot );
  pViewer->setGLRenderAction(
    new SoBoxHighlightRenderAction );
...
```

After clicking:



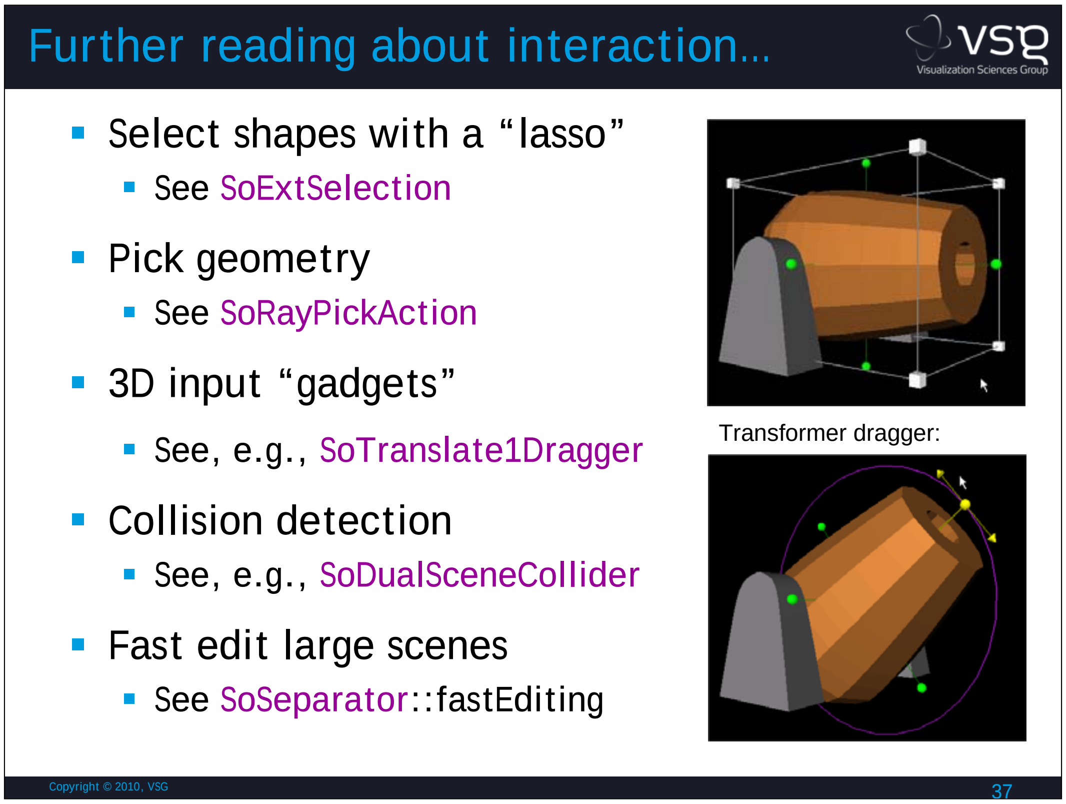
Let’s modify our example program to handle selection and do automatic highlighting. Just for fun I’ve loaded a more complex and interesting scene, but remember it’s just a collection of property and shape nodes even though they’re dressed up with fancy texturing and an image background.

The first step is to create the root node as an **SoSelection** instead of an SoSeparator. Next we’ll specify the functions to be called when an entity is selected or deselected. We won’t go into what those functions actually do. It doesn’t really matter for this example if they do anything at all. These few lines of code are all that’s required to implement high level entity selection.

Creating the viewer and setting the scene graph is the same as usual.

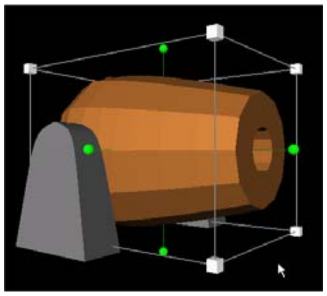
Now we activate automatic highlighting. First we tell the viewer to schedule a redraw whenever the selection list of the specified SoSelection node changes. This is necessary because the selection list is not a *field* of SoSelection and therefore does not automatically trigger a redraw. Normally the viewer does rendering by applying an SoGLRenderAction, which it creates for us, to the scene graph. Now we will tell the viewer to use a special subclass of SoGLRenderAction, called **SoBoxHighlightRenderAction**. This action renders the scene graph as usual, then computes the bounding box of each entity in the selection list and renders it as a wireframe box.

If we click on the front-most column in the scene, the central cylinder of that column is highlighted.

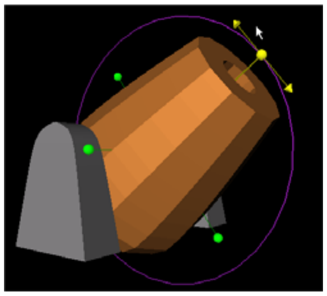


Further reading about interaction…

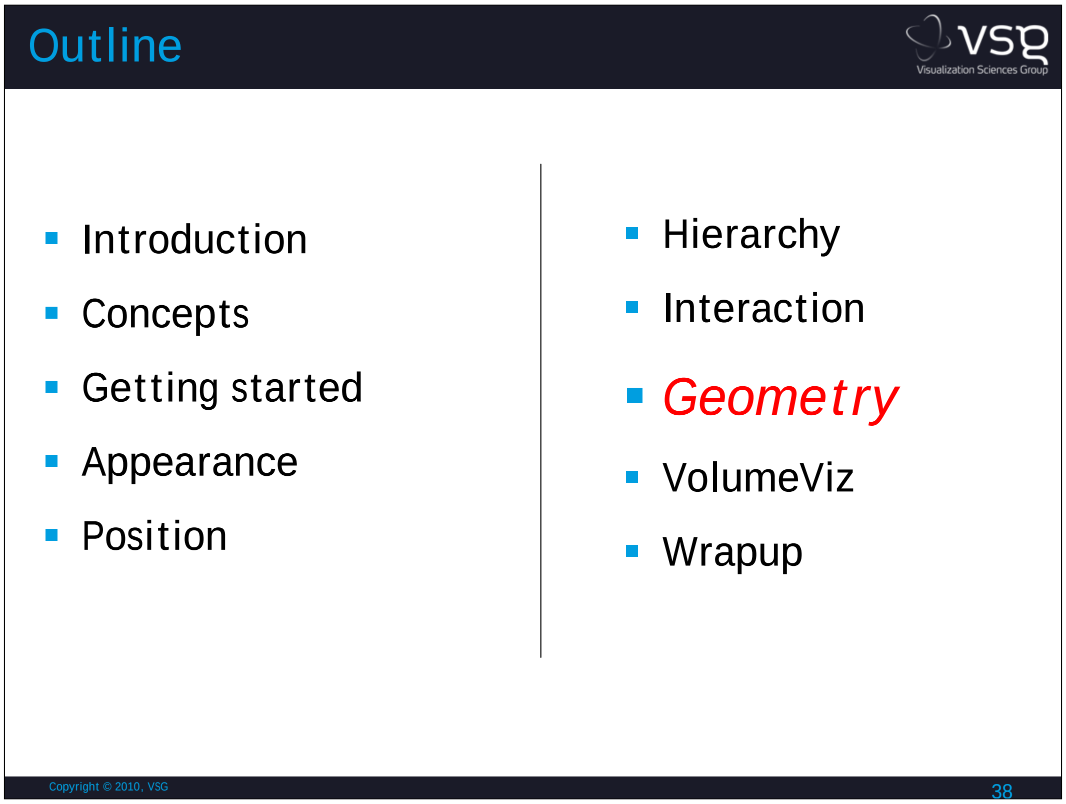
- Select shapes with a "lasso"
  - See SoExtSelection
- Pick geometry
  - See SoRayPickAction
- 3D input "gadgets"
  - See, e.g., SoTranslate1Dragger
- Collision detection
  - See, e.g., SoDualSceneCollider
- Fast edit large scenes
  - See SoSeparator::fastEditing



Transformer dragger:



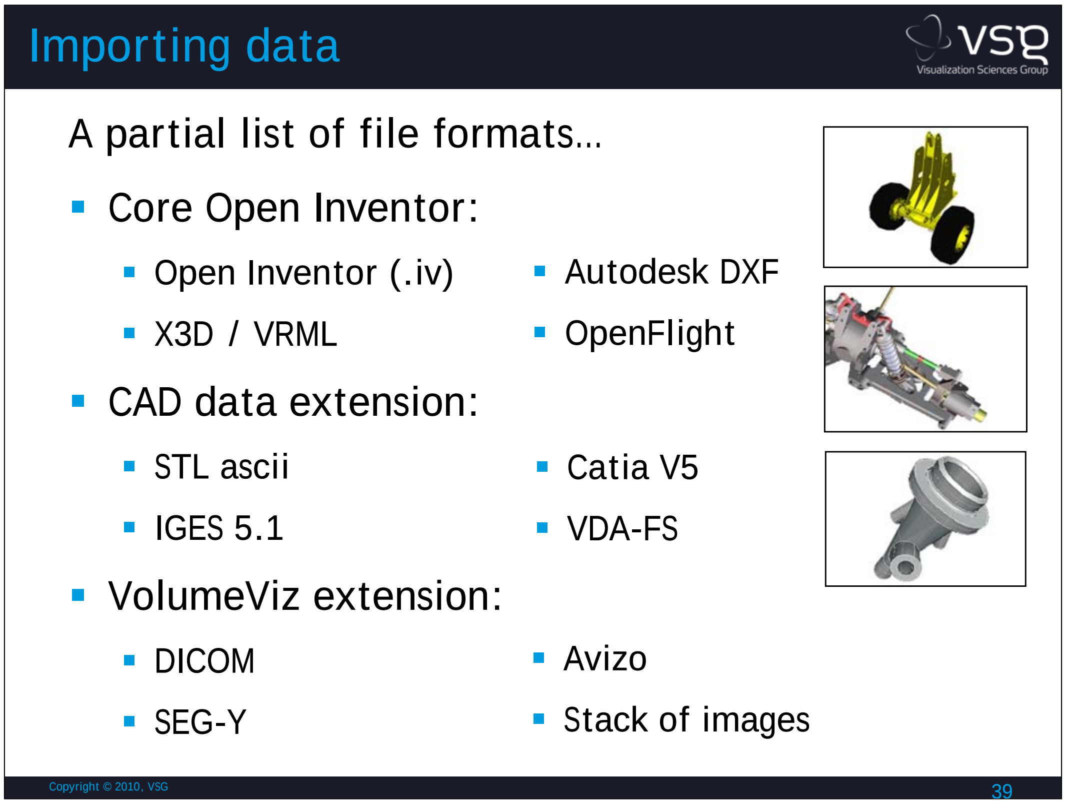
This short example just "scratches the surface" of the features provided for interaction. Here are just a few of the other things you may want to take advantage in your application. You can find more information about these features in the Open Inventor documentation. We will come back to the Translate1Dragger later and see a specific case where this input gadget is very useful.



Outline

- Introduction
- Concepts
- Getting started
- Appearance
- Position
- Hierarchy
- Interaction
- *Geometry*
- VolumeViz
- Wrapup

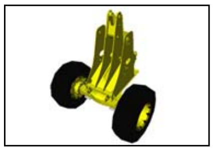
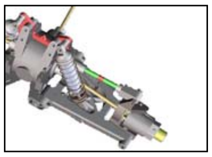
Now we have a 3D visualization program that's interactive, but we need some "real" geometry in the scene. How do we do that?



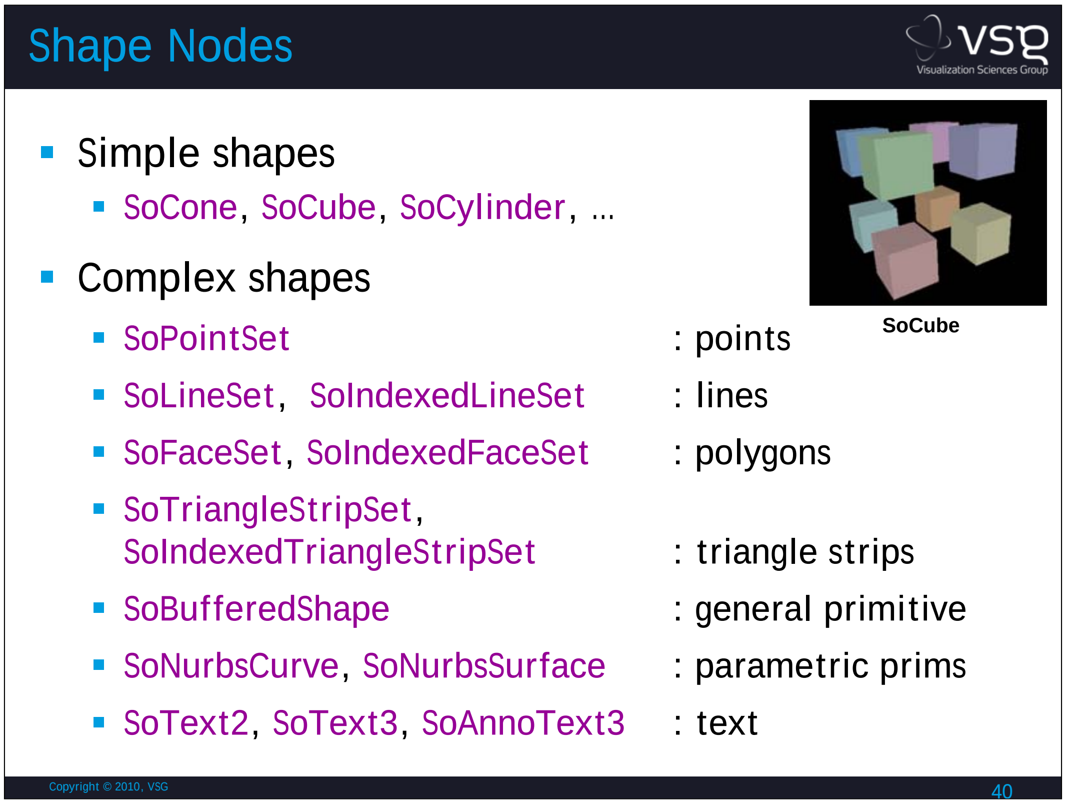
Importing data

A partial list of file formats…

- Core Open Inventor:
  - Open Inventor (.iv)
  - X3D / VRML
  - Autodesk DXF
  - OpenFlight
- CAD data extension:
  - STL ascii
  - IGES 5.1
  - Catia V5
  - VDA-FS
- VolumeViz extension:
  - DICOM
  - SEG-Y
  - Avizo
  - Stack of images

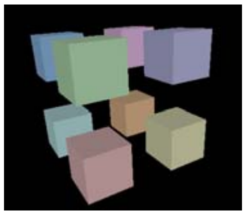



For some applications it's important to be able to import data from another application or recording format. Some applications will prefer to handle their own data import or license this capability from third party specialists, but Open Inventor converts a variety of file formats directly into a scene graph in memory. This can at least be very valuable for prototyping. I won't read through or explain what all these formats are. For more information see the documentation, particularly the **SoInput** class for geometric data and the **SoVolumeData** class for volume data. Next we'll talk about creating your own geometry in Open Inventor.



Shape Nodes

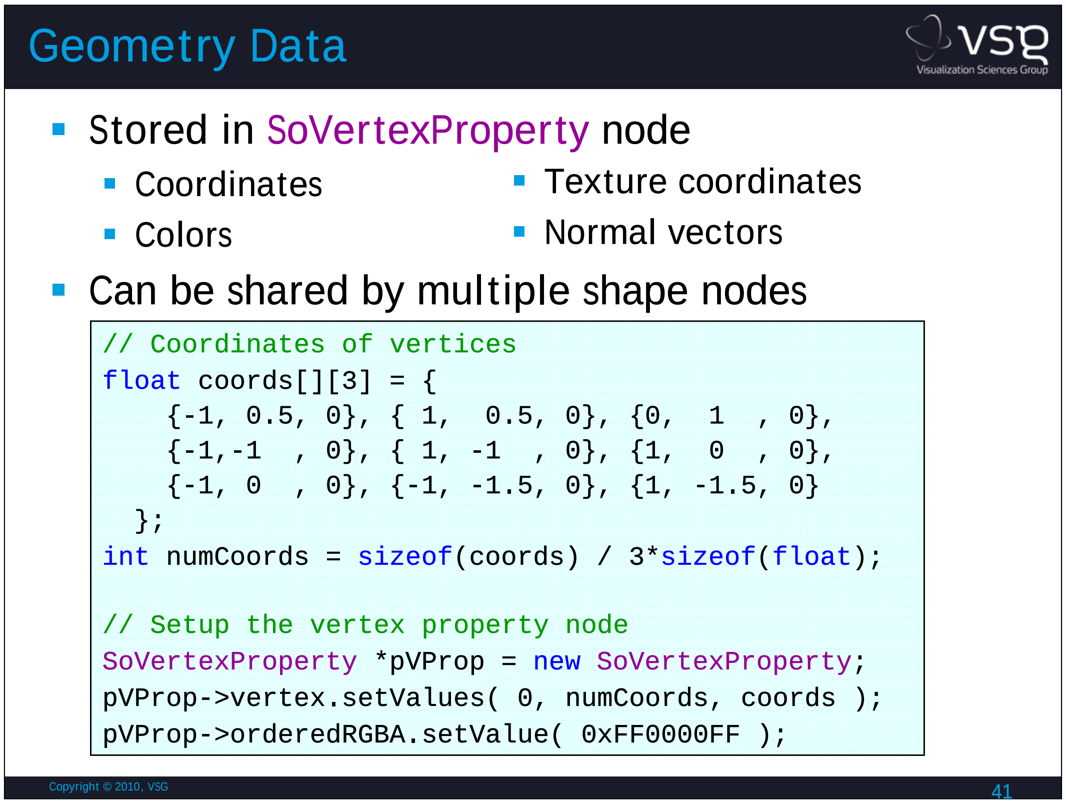
- Simple shapes
  - SoCone, SoCube, SoCylinder, …
- Complex shapes
  - SoPointSet : points
  - SoLineSet, SoIndexedLineSet : lines
  - SoFaceSet, SoIndexedFaceSet : polygons
  - SoTriangleStripSet, SoIndexedTriangleStripSet : triangle strips
  - SoBufferedShape : general primitive
  - SoNurbsCurve, SoNurbsSurface : parametric prims
  - SoText2, SoText3, SoAnnoText3 : text


SoCube

40

There are many different kinds of shape nodes in Open Inventor. We've already met some of the simple predefined shapes like Cone, Cube and Cylinder.

<u><CLICK!></u> We also have nodes specifically for rendering text, points, lines, polygons, triangle strips, NURBS surfaces and much more.



Geometry Data

- Stored in SoVertexProperty node
  - Coordinates
  - Colors
  - Texture coordinates
  - Normal vectors
- Can be shared by multiple shape nodes

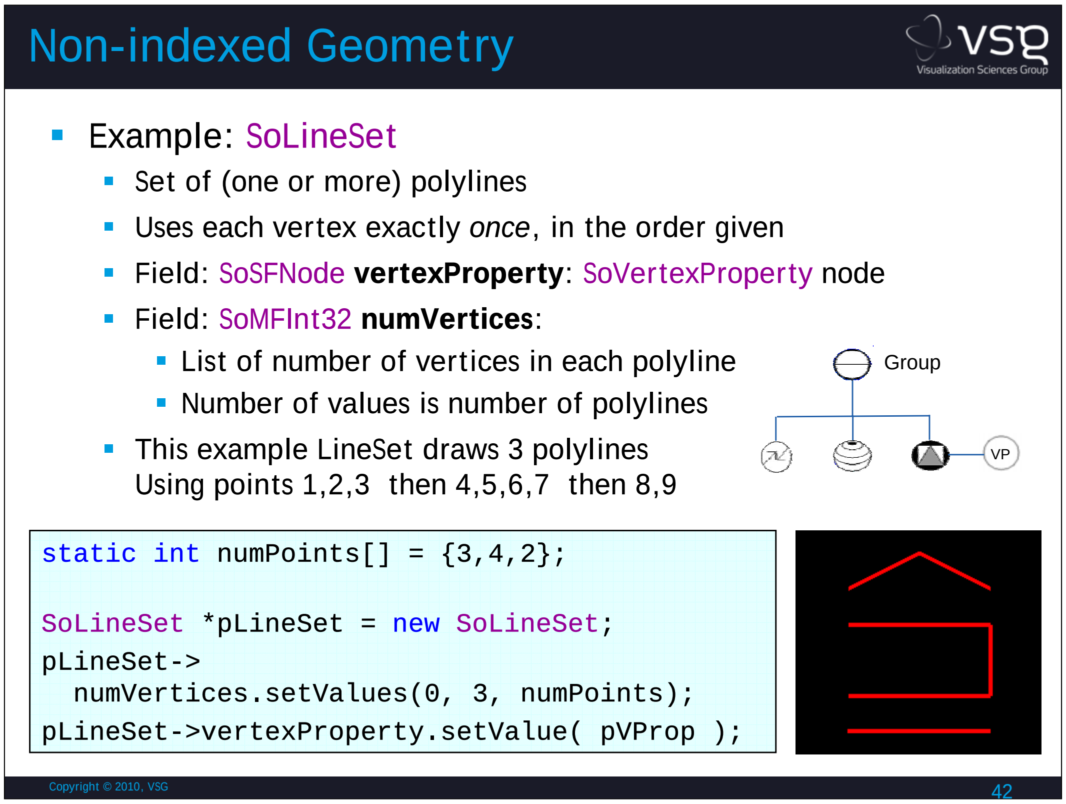
```
// Coordinates of vertices
float coords[][3] = {
    {-1, 0.5, 0}, { 1,  0.5, 0}, {0,  1  , 0},
    {-1,-1  , 0}, { 1, -1  , 0}, {1,  0  , 0},
    {-1, 0  , 0}, {-1, -1.5, 0}, {1, -1.5, 0}
  };
int numCoords = sizeof(coords) / 3*sizeof(float);

// Setup the vertex property node
SoVertexProperty *pVProp = new SoVertexProperty;
pVProp->vertex.setValues( 0, numCoords, coords );
pVProp->orderedRGBA.setValue( 0xFF0000FF );
```

The list of coordinates, and other data, that define a shape's vertices is normally stored in the scene graph using an **SoVertexProperty** node. In addition to coordinates and colors, SoVertexProperty can also store normal vectors and texture coordinates, which we don't have time to discuss today.

<u><CLICK!></u> Coordinates are stored in the *vertex* field. One way to set the coordinate values is set all the values in one call to the *vertex* field's *setValues*() method. This method also allows us to replace a subset of the existing values in the field because it takes both a starting index and a count of values to store. It's also possible to store the values one at a time using the *set1Value()* method, but this is very inefficient for large coordinate arrays.

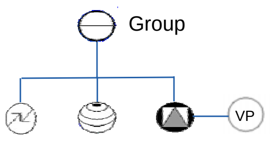
Color and opacity are stored in the *orderedRGBA* field. For now we're happy with making the whole box red, so we only have to set a single color value in the *orderedRGBA* field. We're setting the diffuseColor, but note that unlike the SoMaterial node, the color is a packed RGBA value. In other words an unsigned 32-bit integer with 8 bits each of red, green, blue and alpha (the inverse of transparency). The value we used here is full red and full opacity.

If a shape needs a very large number of coordinates, it may be even more efficient to use an **SoBufferedShape** node.

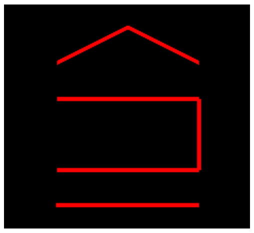


Non-indexed Geometry

- Example: SoLineSet
  - Set of (one or more) polylines
  - Uses each vertex exactly *once*, in the order given
  - Field: SoSFNode **vertexProperty**: SoVertexProperty node
  - Field: SoMFInt32 **numVertices**:
    - List of number of vertices in each polyline
    - Number of values is number of polylines
  - This example LineSet draws 3 polylines
    Using points 1,2,3 then 4,5,6,7 then 8,9



```
static int numPoints[] = {3,4,2};

SoLineSet *pLineSet = new SoLineSet;
pLineSet->
  numVertices.setValues(0, 3, numPoints);
pLineSet->vertexProperty.setValue( pVProp );
```

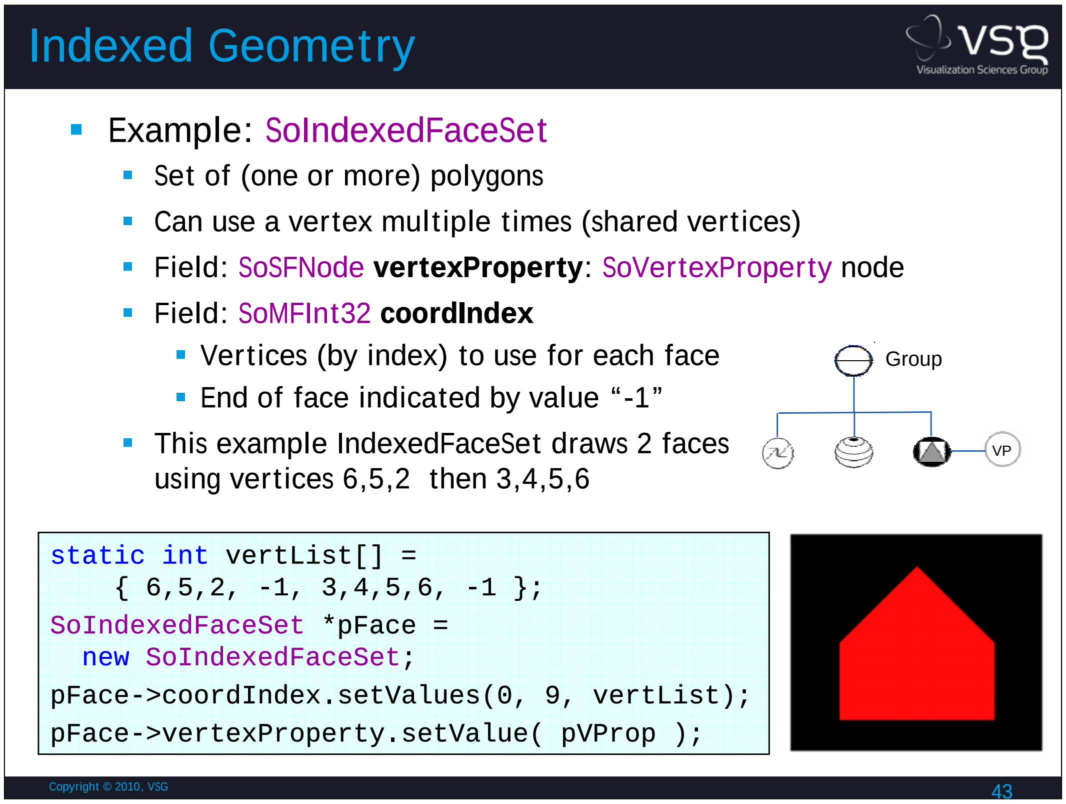


The geometry nodes that use a list of coordinates are divided into *indexed* and *non-indexed* primitives. We'll look at the non-indexed case first because it's simpler. Non-indexed means that the geometry node uses the list of coordinates in order and uses each coordinate once. It doesn't necessarily use *all* the coordinates. This is a convenient way to define a primitive if none of the vertices are "shared", meaning used more than once in the definition of the primitive.

<u><CLICK!></u> We'll use the **SoVertexProperty** node, named "pVProp" that we created on the previous slide. SoVertexProperty is efficient because it encapsulates all the lists a geometry might need, including normal vectors, texture coordinates, colors, etc. All the vertex based geometry nodes have a *vertexProperty* field. This is an *SoSFNode* field, a special type of field that contains another node. In this case an SoVertexProperty node.

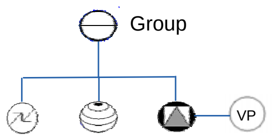
<u><CLICK!></u> Let's look at **SoLineSet**. This is a non-indexed primitive that draws one or more polylines. A polyline is a line made up of one or more segments. SoLineSet, like all the other non-indexed geometry nodes, has a multi-valued integer field *numVertices* that specifies the number of polylines and how many vertices should be used for each polyline.

<u><CLICK!></u> In this case we want to draw 3 polylines, so we put 3 values in the *numVertices* field. The first line is drawn using the first 3 vertices, the second line is drawn using the next 4 vertices and the third line is drawn using the next 2 vertices.



Indexed Geometry

- Example: SoIndexedFaceSet
  - Set of (one or more) polygons
  - Can use a vertex multiple times (shared vertices)
  - Field: SoSFNode **vertexProperty**: SoVertexProperty node
  - Field: SoMFInt32 **coordIndex**
    - Vertices (by index) to use for each face
    - End of face indicated by value “-1”
  - This example IndexedFaceSet draws 2 faces using vertices 6,5,2 then 3,4,5,6



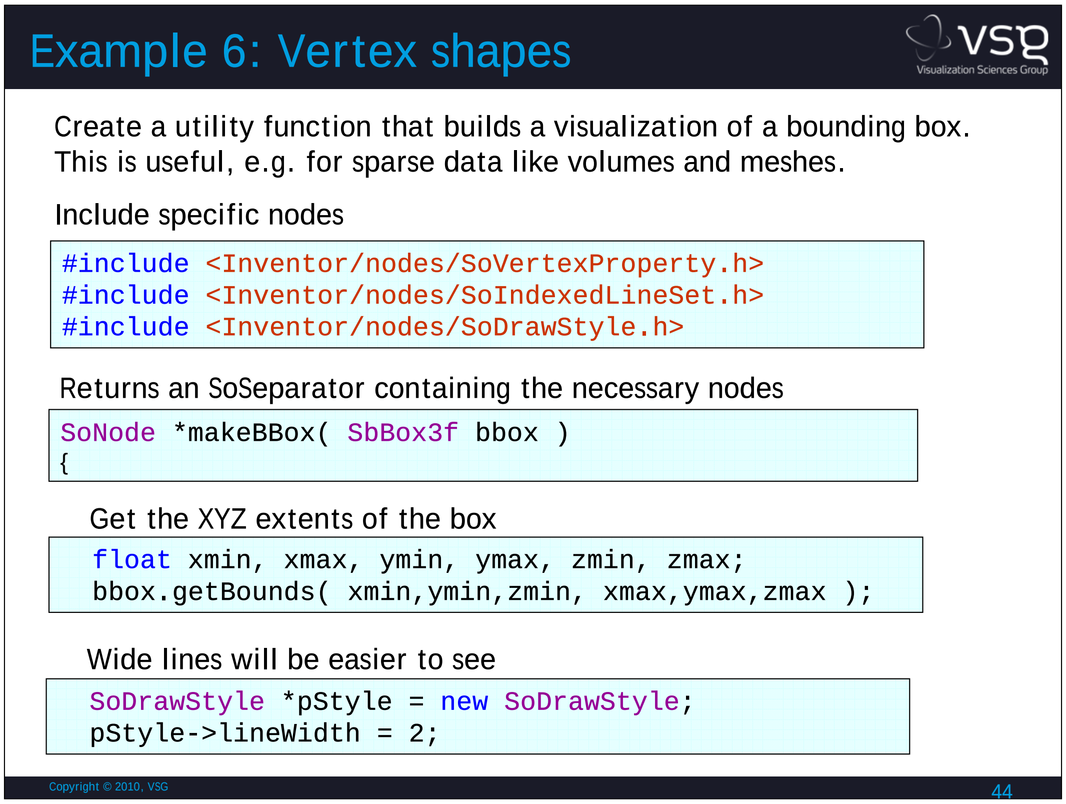
```
static int vertList[] =
    { 6,5,2, -1, 3,4,5,6, -1 };
SoIndexedFaceSet *pFace =
  new SoIndexedFaceSet;
pFace->coordIndex.setValues(0, 9, vertList);
pFace->vertexProperty.setValue( pVProp );
```

The indexed primitives are more general. They have the advantage that vertices can be re-used, because each polyline or face in the primitive is defined by a list of integer indices specifying which vertex in the list to use. For example in a surface mesh, four or more faces can have a common, shared vertex. Putting each coordinate in the vertex list once can save a lot of memory in this case.

<CLICK!> Just like the previous example, we’ll put our SoVertexProperty node containing the coordinates into the vertexProperty field.

<CLICK!> Let’s look at **SoIndexedFaceSet**. This is an indexed primitive that draws one or more polygons or faces. Like the other indexed geometry nodes, it has a multi-valued integer field *coordIndex* that specifies the number of faces and which vertices should be used for each face.

<CLICK!> This **SoIndexedFaceSet** is going to draw 2 polygons. The first one has 3 vertices (a triangle) and the second one has 4 vertices (a rectangle).   The faces share vertices 5 and 6. We don’t have to specify the number of indices in each face. Instead we mark the end of each face with the special index value -1. As you can see, the polygon primitives are very general. There is no limit (except memory) on the number of vertices and unlike OpenGL, Open Inventor supports both convex and concave polygons, and even polygons with “holes”. We don’t have time to talk about that today, but there is a good discussion in the User’s Guide .


Example 6: Vertex shapes

Create a utility function that builds a visualization of a bounding box.
This is useful, e.g. for sparse data like volumes and meshes.

Include specific nodes

```
#include <Inventor/nodes/SoVertexProperty.h>
#include <Inventor/nodes/SoIndexedLineSet.h>
#include <Inventor/nodes/SoDrawStyle.h>
```

Returns an SoSeparator containing the necessary nodes

```
SoNode *makeBBox( SbBox3f bbox )
{
```

Get the XYZ extents of the box

```
  float xmin, xmax, ymin, ymax, zmin, zmax;
  bbox.getBounds( xmin,ymin,zmin, xmax,ymax,zmax );
```

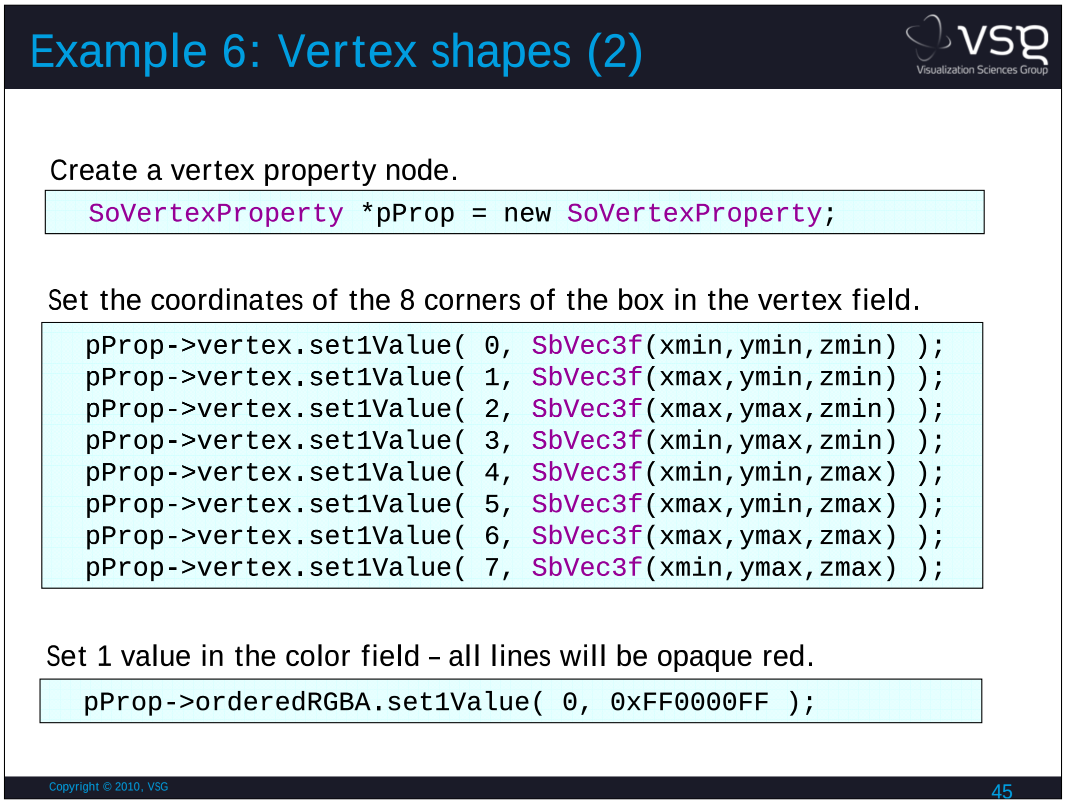
Wide lines will be easier to see

```
  SoDrawStyle *pStyle = new SoDrawStyle;
  pStyle->lineWidth = 2;
```

Let’s walk through a complete example of creating geometry. Suppose that we need to draw the outline of a bounding box enclosing some region of 3D space. This is a very common feature in applications that are visualizing sparse data like volumes and meshes. To do that we’ll need a collection of Open Inventor nodes including **SoVertexProperty**, **SoLineSet**, some property nodes and an **SoSeparator** to group them together. Let’s create a utility function that takes the extent of the box as an **SbBox3f** and returns an SoSeparator node containing the complete entity. Note that there are more elegant, object oriented ways to do this in Open Inventor, for example creating a new type of node kit, but we don’t have time to discuss that today.

An SbBox3f is defined by the min and max values along each axis. First we’ll get those values to use as the corner coordinates of our box.

Next we’ll create an SoDrawStyle node so we can set the width of the lines used to draw the box .



Example 6: Vertex shapes (2)

Create a vertex property node.

```
   SoVertexProperty *pProp = new SoVertexProperty;
```

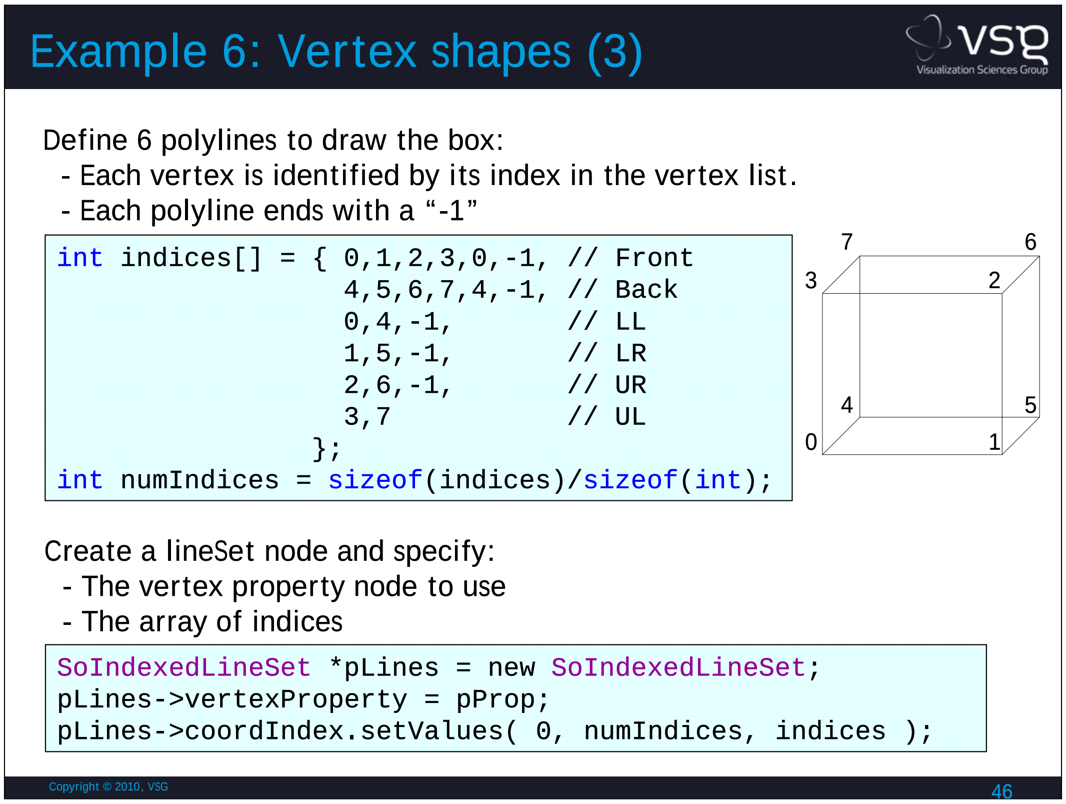
Set the coordinates of the 8 corners of the box in the vertex field.

```
   pProp->vertex.set1Value( 0, SbVec3f(xmin,ymin,zmin) );
   pProp->vertex.set1Value( 1, SbVec3f(xmax,ymin,zmin) );
   pProp->vertex.set1Value( 2, SbVec3f(xmax,ymax,zmin) );
   pProp->vertex.set1Value( 3, SbVec3f(xmin,ymax,zmin) );
   pProp->vertex.set1Value( 4, SbVec3f(xmin,ymin,zmax) );
   pProp->vertex.set1Value( 5, SbVec3f(xmax,ymin,zmax) );
   pProp->vertex.set1Value( 6, SbVec3f(xmax,ymax,zmax) );
   pProp->vertex.set1Value( 7, SbVec3f(xmin,ymax,zmax) );
```

Set 1 value in the color field – all lines will be opaque red.

```
   pProp->orderedRGBA.set1Value( 0, 0xFF0000FF );
```

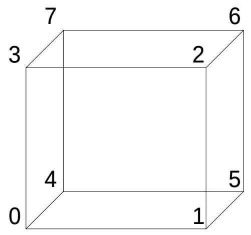
Next we'll create an SoVertexProperty node to hold the coordinates. We don't have the coordinates in an array like we did in the previous example, so we'll poke them into the *vertex* field one at a time. There are 8 coordinates defining the corners of the box and the index numbers we assign here are the values we'll use in the geometry node. As in the previous example, our box will be opaque red. However orderedRGBA is a multi-value field so we can assign a different color to each face, a different color to each vertex and so on.



Example 6: Vertex shapes (3)

Define 6 polylines to draw the box:
- Each vertex is identified by its index in the vertex list.
- Each polyline ends with a "-1"

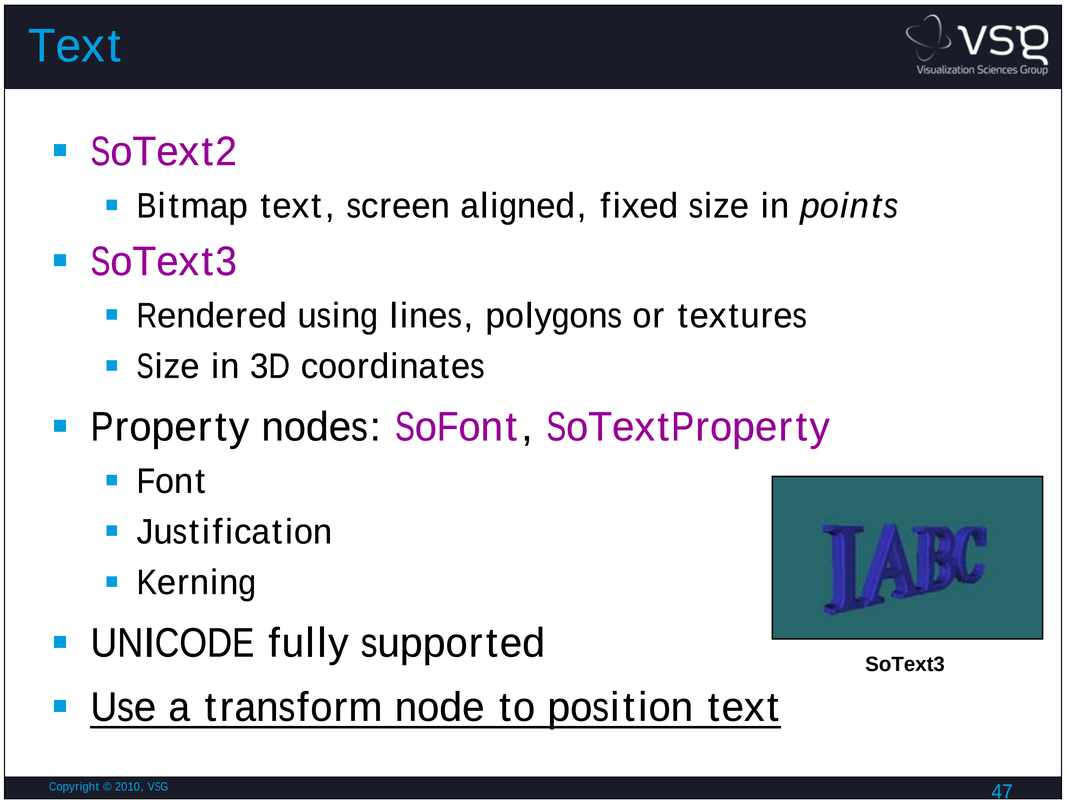
```
int indices[] = { 0,1,2,3,0,-1, // Front
                  4,5,6,7,4,-1, // Back
                  0,4,-1,       // LL
                  1,5,-1,       // LR
                  2,6,-1,       // UR
                  3,7           // UL
                };
int numIndices = sizeof(indices)/sizeof(int);
```



Create a lineSet node and specify:
- The vertex property node to use
- The array of indices

```
SoIndexedLineSet *pLines = new SoIndexedLineSet;
pLines->vertexProperty = pProp;
pLines->coordIndex.setValues( 0, numIndices, indices );
```

It can be helpful to make a diagram and number the vertices in the primitive, so it's easier to get the right index values in the right place in the index list. There are multiple ways to draw the box. To avoid retracing any edges we'll use 6 polylines, one for the front, one for the back and one each for the connecting edges. Notice how each polyline is terminated by the special -1 value.



Text

- SoText2
  - Bitmap text, screen aligned, fixed size in *points*
- SoText3
  - Rendered using lines, polygons or textures
  - Size in 3D coordinates
- Property nodes: SoFont, SoTextProperty
  - Font
  - Justification
  - Kerning
- UNICODE fully supported
- Use a transform node to position text

SoText3

Copyright © 2010, VSG 47

Text is very important in some applications for annotating the 3D scene. There are multiple text nodes in Open Inventor, but the two main classes are SoText2 and SoText3.

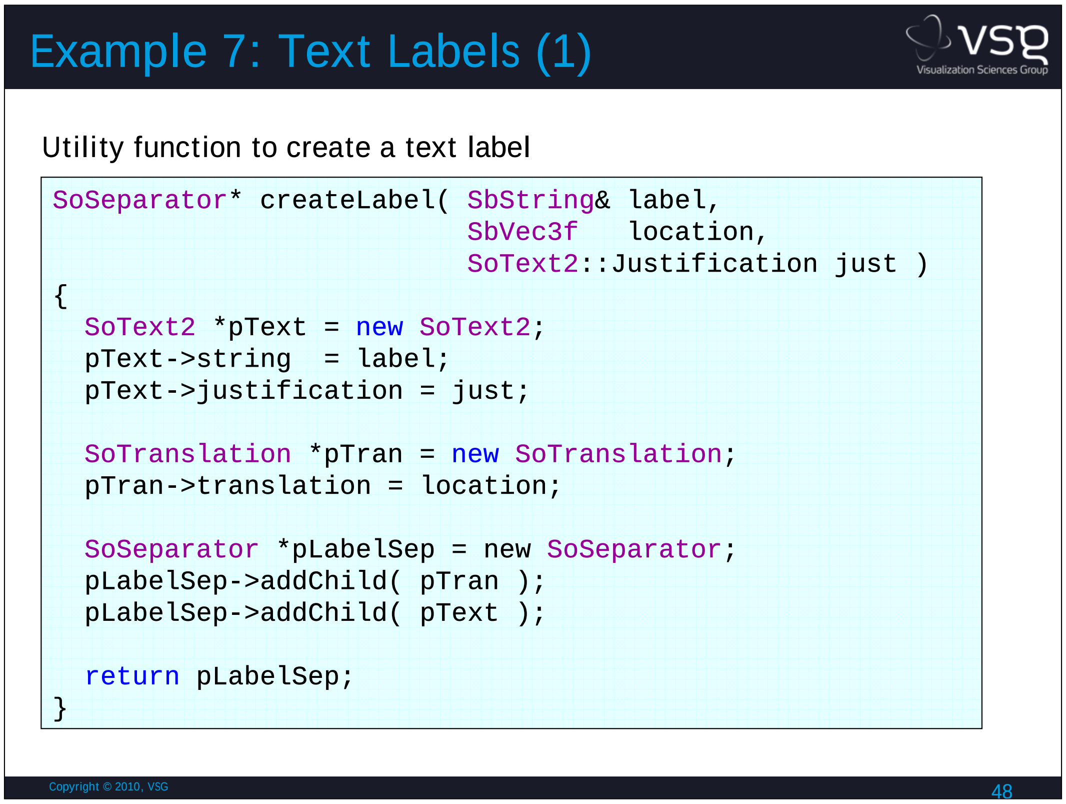
•**SoText2** is drawn using bitmaps, is always screen aligned and has a constant physical size on screen specified in *points*. So the apparent size of the text does not change as the camera moves closer to, or farther away from, the scene. This feature may or may not be desirable. Having the text always facing the screen can be useful.

<CLICK!>

•**SoText3** is rendered using lines, polygons or textures, depending on the setting in the SoFont node. The size is specified in 3D units. The text is drawn in the XY plane and rotates with the scene.

<CLICK!> All the text nodes contain the text to be rendered and a few basic properties. The property nodes **SoFont** and **SoTextProperty** specify the font to be used, size, justification and other properties. Note that all the text nodes support multiple lines of text and also support Unicode strings, allowing full internationalization of your application.

<CLICK!> It's very important to note what the text nodes *don't* include, namely a position. Similar to the simple shape nodes like SoCone, text is always rendered at 0,0,0. So we'll normally need an **SoTransform** or **SoTranslation** node to move the text to the desired location. Of course for 3D text we can also use transform nodes to rotate the strings.



Example 7: Text Labels (1)

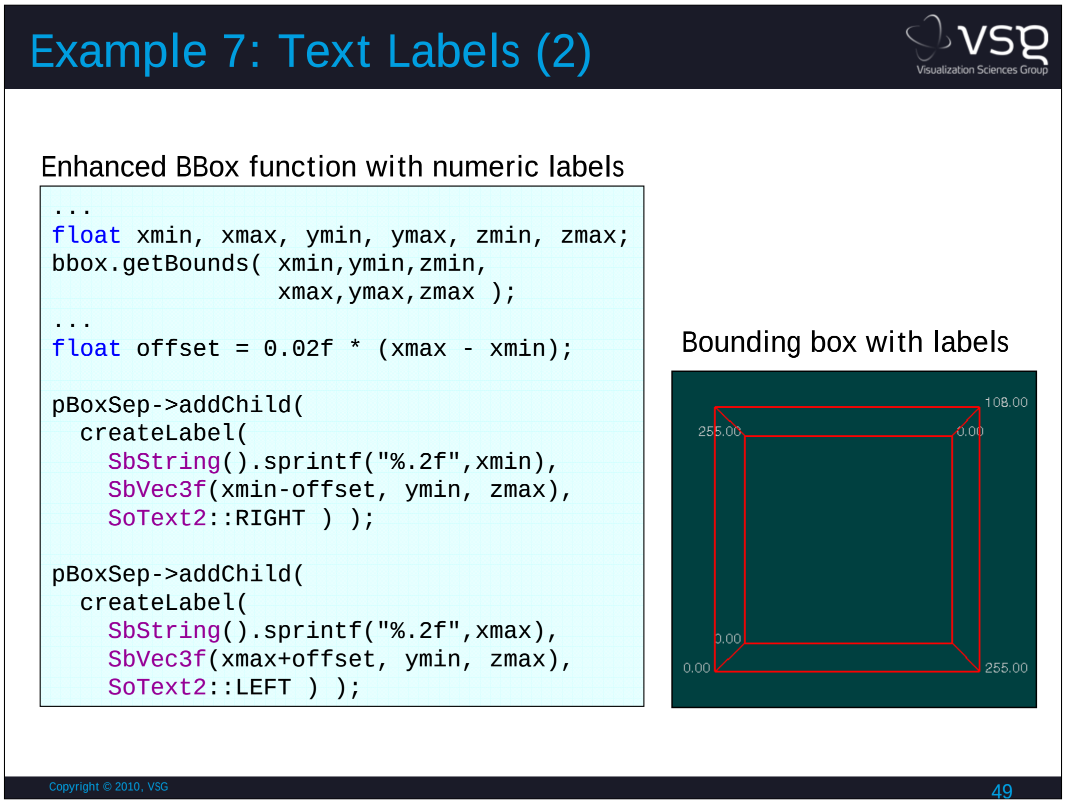
Utility function to create a text label

```
SoSeparator* createLabel( SbString& label,
                          SbVec3f   location,
                          SoText2::Justification just )
{
  SoText2 *pText = new SoText2;
  pText->string  = label;
  pText->justification = just;

  SoTranslation *pTran = new SoTranslation;
  pTran->translation = location;

  SoSeparator *pLabelSep = new SoSeparator;
  pLabelSep->addChild( pTran );
  pLabelSep->addChild( pText );

  return pLabelSep;
}
```

As an example let’s create a utility function to place some 2D (bitmap, screen aligned) text at a specified location. The label string is specified using Open Inventor’s string class, but you can also use char*, std::string, and so on. The label location is a 3D point specified using Open Inventor’s vector class. And the horizontal justification is specified using the enum type in SoText2.

The implementation of this function should be easy to read by now. We create the text node and specify the text to be rendered and the horizontal justification. We create a translation node to position the text. Finally we create a Separator to hold the nodes and prevent the translation from affecting other nodes in the scene graph. Note that we created the text node first, but we put the translation node into the scene graph first so that it will affect the text node.

We might need to create other property nodes in this function, for example to set the font and size. However typically all, or at least many, strings use the same font and size, so it’s more convenient to place the **SoFont** node higher in the scene graph so its values are inherited by all the text nodes.

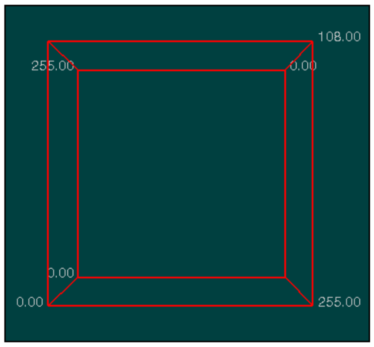


Example 7: Text Labels (2)

Enhanced BBox function with numeric labels

```
...
float xmin, xmax, ymin, ymax, zmin, zmax;
bbox.getBounds( xmin,ymin,zmin,
                xmax,ymax,zmax );
...
float offset = 0.02f * (xmax - xmin);

pBoxSep->addChild(
  createLabel(
    SbString().sprintf("%.2f",xmin),
    SbVec3f(xmin-offset, ymin, zmax),
    SoText2::RIGHT ) );

pBoxSep->addChild(
  createLabel(
    SbString().sprintf("%.2f",xmax),
    SbVec3f(xmax+offset, ymin, zmax),
    SoText2::LEFT ) );
```

Bounding box with labels



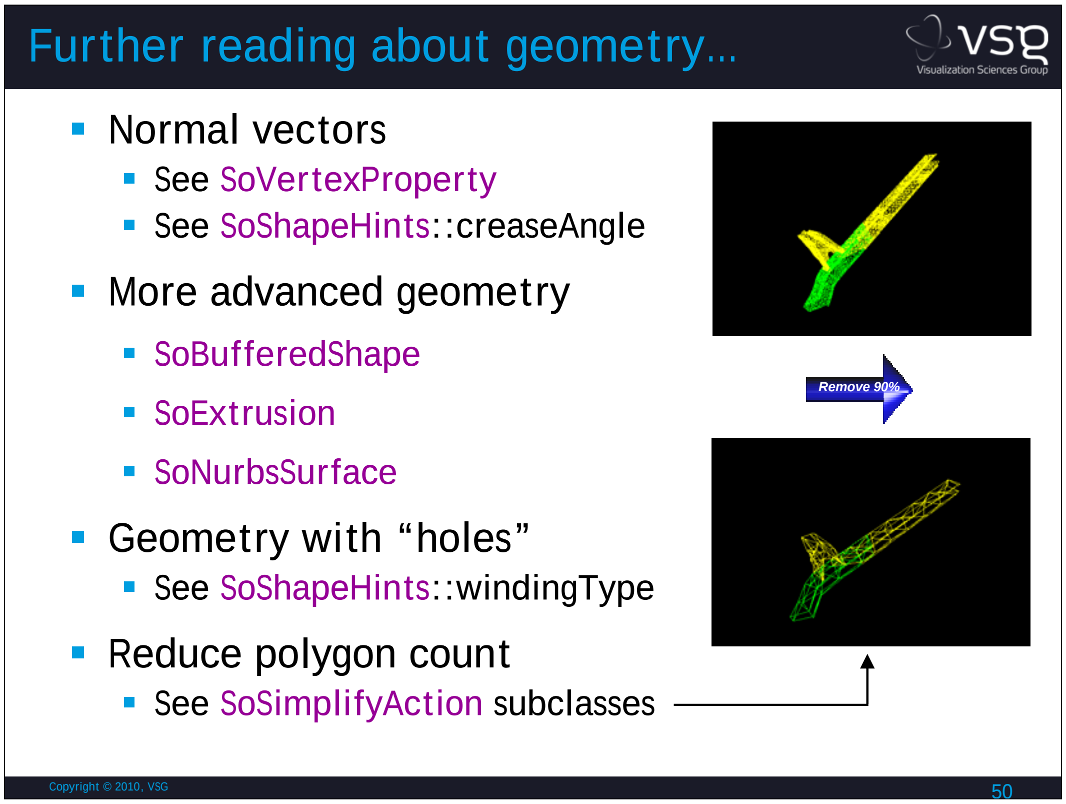
Copyright © 2010, VSG

49

Now we can use our new utility function in the makeBBox function to add labels to the bounding box, taking advantage of the left or right justification option. We'll probably want a small offset to move the text away from the lines. One way to compute a reasonable offset is as a small percentage of the extent of the box along each axis. Here we'll just show the code for the X axis since the other axes are simple variations. We're going to draw the X axis labels on the bottom-front edge of the box. The coordinate for the Xmin value will be xmin-offset, ymin, zmax. Ymin because Y is up. Zmax because the default Z axis comes out of the screen, so larger Z values are toward the "front". The Xmin label will use RIGHT justification, meaning that Open Inventor will automatically shift the text string so that its lower right corner is at the specified coordinate. The Xmax label is handled similarly, except the position is offset to the right from Xmax and LEFT justification is used.

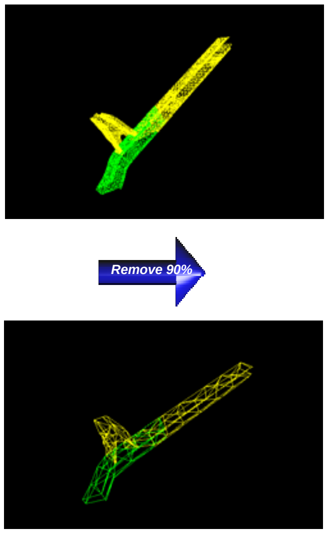
Note that, just for convenience, we used the **SbString** class's sprintf method to construct a string from the Xmin and Xmax values.

<CLICK!> On the right you can see the resulting image on the screen, for an arbitrary bounding box.



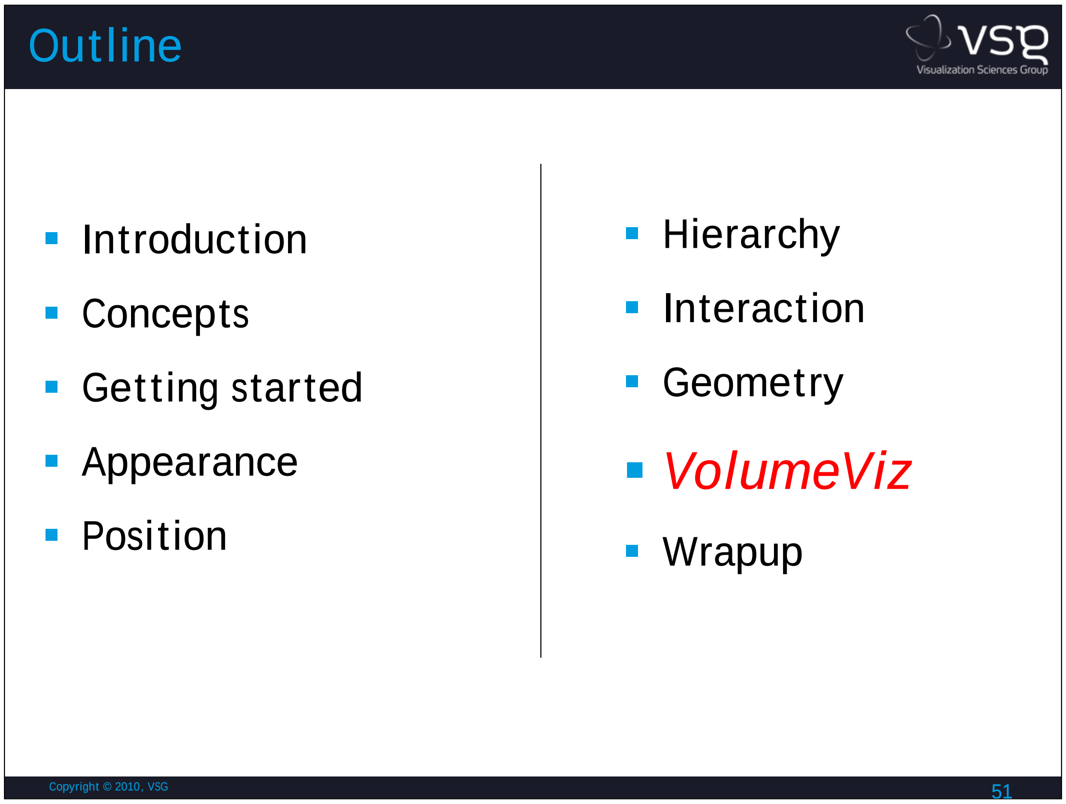
Further reading about geometry…

- Normal vectors
  - See SoVertexProperty
  - See SoShapeHints::creaseAngle
- More advanced geometry
  - SoBufferedShape
  - SoExtrusion
  - SoNurbsSurface
- Geometry with "holes"
  - See SoShapeHints::windingType
- Reduce polygon count
  - See SoSimplifyAction subclasses



50

This short example just "scratches the surface" of the features provided by the shape classes. Here are just a few of the other things you may want to take advantage in your application. You can find more information about these features in the Open Inventor documentation. In particular the simplify action can be used to intelligently reduce the polygon count of an object, for example to create levels of detail.



Outline

- Introduction
- Concepts
- Getting started
- Appearance
- Position
- Hierarchy
- Interaction
- Geometry
- *VolumeViz*
- Wrapup

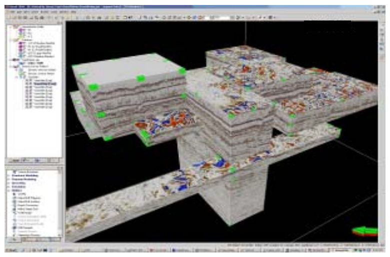
51

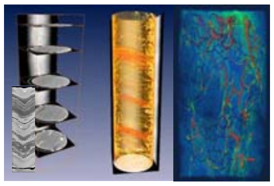
Our wireframe box with text labels would be very handy if we needed to display, for example, volume data. How could we do that?

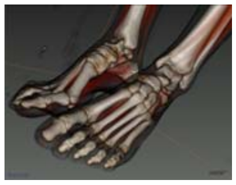
VolumeViz

A toolkit for managing & visualizing volume data

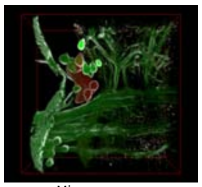
- Rendering tools
  - OrthoSlice / ObliqueSlice / FenceSlice
  - Volume Rendering / Volume Skin
  - Volume Geometry
- High performance
- High quality rendering
- Extensible shader framework
- Data management


Seismic data volume rendering, courtesy Schlumberger


Material science

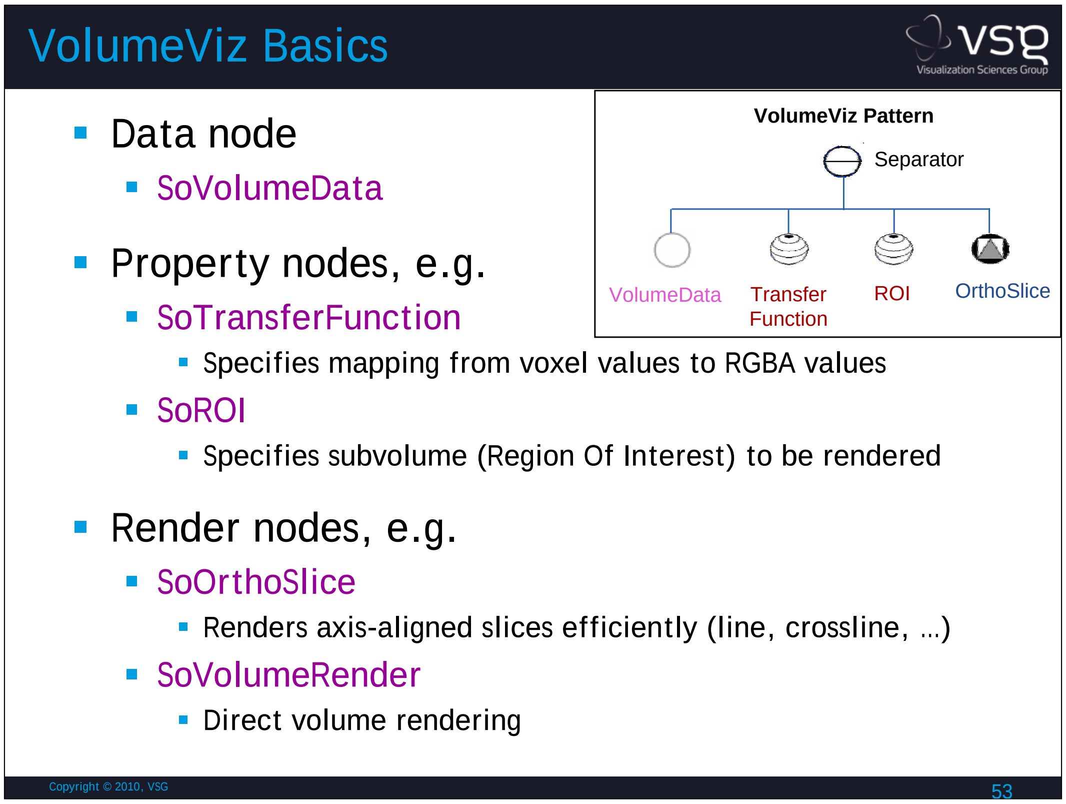

Medical

Medical


Microscopy

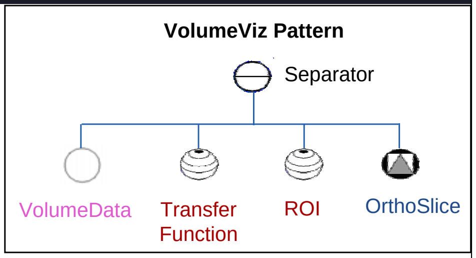
As I mentioned in the beginning of the presentation, the VolumeViz extension adds additional nodes specifically designed for managing and visualizing volume data, using a variety of techniques.

Here we see examples of volume data from seismic, material science, medical and microscopy applications.



VolumeViz Basics

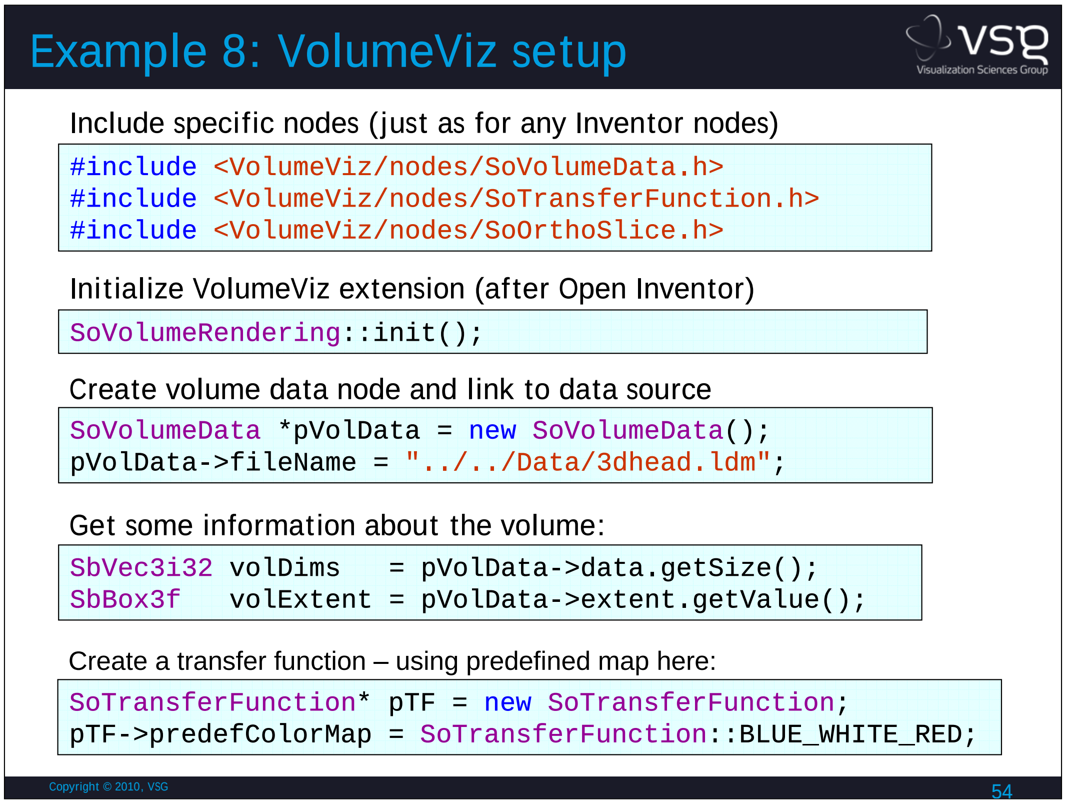
- Data node
  - SoVolumeData
- Property nodes, e.g.
  - SoTransferFunction
    - Specifies mapping from voxel values to RGBA values
  - SoROI
    - Specifies subvolume (Region Of Interest) to be rendered
- Render nodes, e.g.
  - SoOrthoSlice
    - Renders axis-aligned slices efficiently (line, crossline, …)
  - SoVolumeRender
    - Direct volume rendering



53

VolumeViz has nodes that fall into the familiar categories: Data nodes, Property nodes and rendering nodes.

In addition you can freely mix VolumeViz rendering and core Open Inventor rendering in the same scene.



Example 8: VolumeViz setup

Include specific nodes (just as for any Inventor nodes)

```
#include <VolumeViz/nodes/SoVolumeData.h>
#include <VolumeViz/nodes/SoTransferFunction.h>
#include <VolumeViz/nodes/SoOrthoSlice.h>
```

Initialize VolumeViz extension (after Open Inventor)

```
SoVolumeRendering::init();
```

Create volume data node and link to data source

```
SoVolumeData *pVolData = new SoVolumeData();
pVolData->fileName = "../../Data/3dhead.ldm";
```

Get some information about the volume:

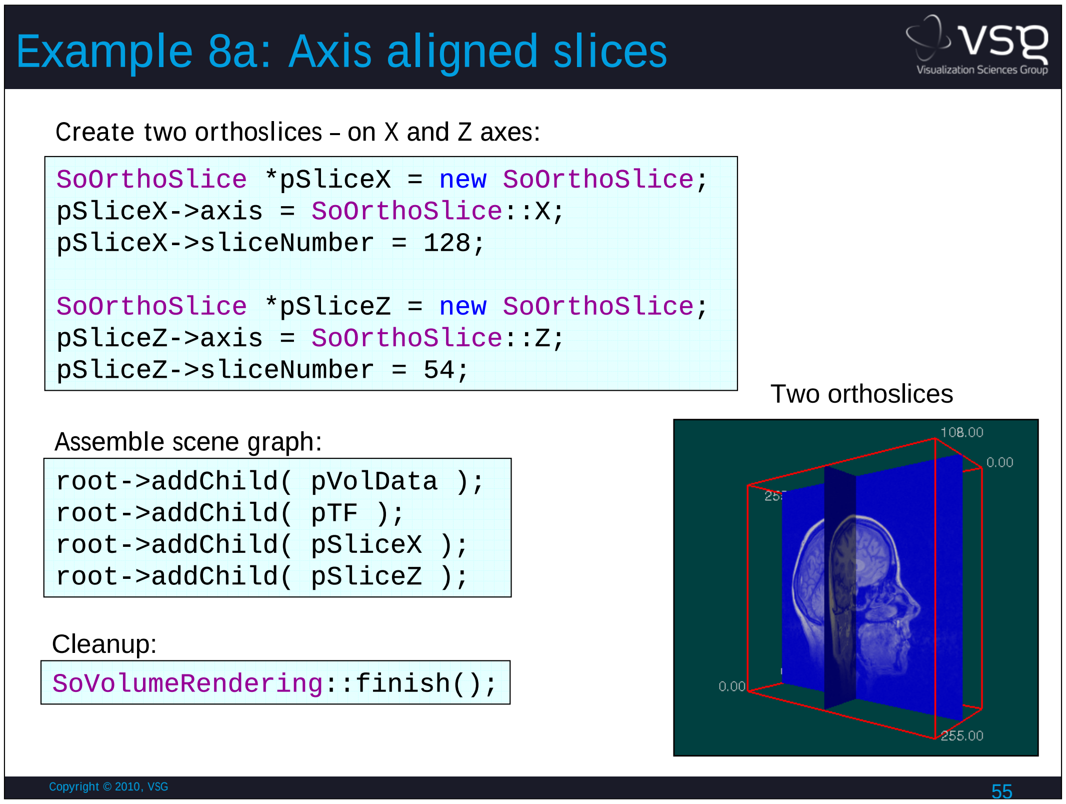
```
SbVec3i32 volDims   = pVolData->data.getSize();
SbBox3f   volExtent = pVolData->extent.getValue();
```

Create a transfer function – using predefined map here:

```
SoTransferFunction* pTF = new SoTransferFunction;
pTF->predefColorMap = SoTransferFunction::BLUE_WHITE_RED;
```

54

The VolumeViz setup is basically the same whether we're rendering slices or doing volume rendering. Note that in the C++ API we have to explicitly initialize the VolumeViz extension. Then we can create a volume data node and specify the file to load the data from. If the file contains a supported data format, here the VolumeViz large data format, VolumeViz will automatically select the appropriate volume reader. As mentioned earlier, applications can create their own volume reader class to interface to different formats or data sources. Using information obtained from the volume reader, SoVolumeData will automatically set the *data*, *extent* and other fields. For illustration purposes we query those values here. Finally we create a transfer function and tell it to use one of the predefined color maps.



Example 8a: Axis aligned slices

Create two orthoslices – on X and Z axes:

```
SoOrthoSlice *pSliceX = new SoOrthoSlice;
pSliceX->axis = SoOrthoSlice::X;
pSliceX->sliceNumber = 128;

SoOrthoSlice *pSliceZ = new SoOrthoSlice;
pSliceZ->axis = SoOrthoSlice::Z;
pSliceZ->sliceNumber = 54;
```

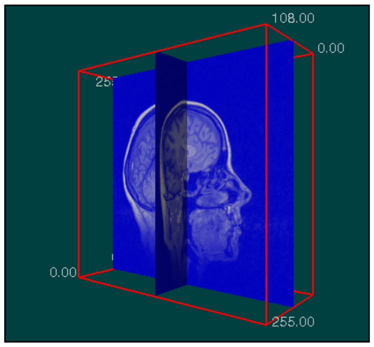
Assemble scene graph:

```
root->addChild( pVolData );
root->addChild( pTF );
root->addChild( pSliceX );
root->addChild( pSliceZ );
```

Cleanup:

```
SoVolumeRendering::finish();
```

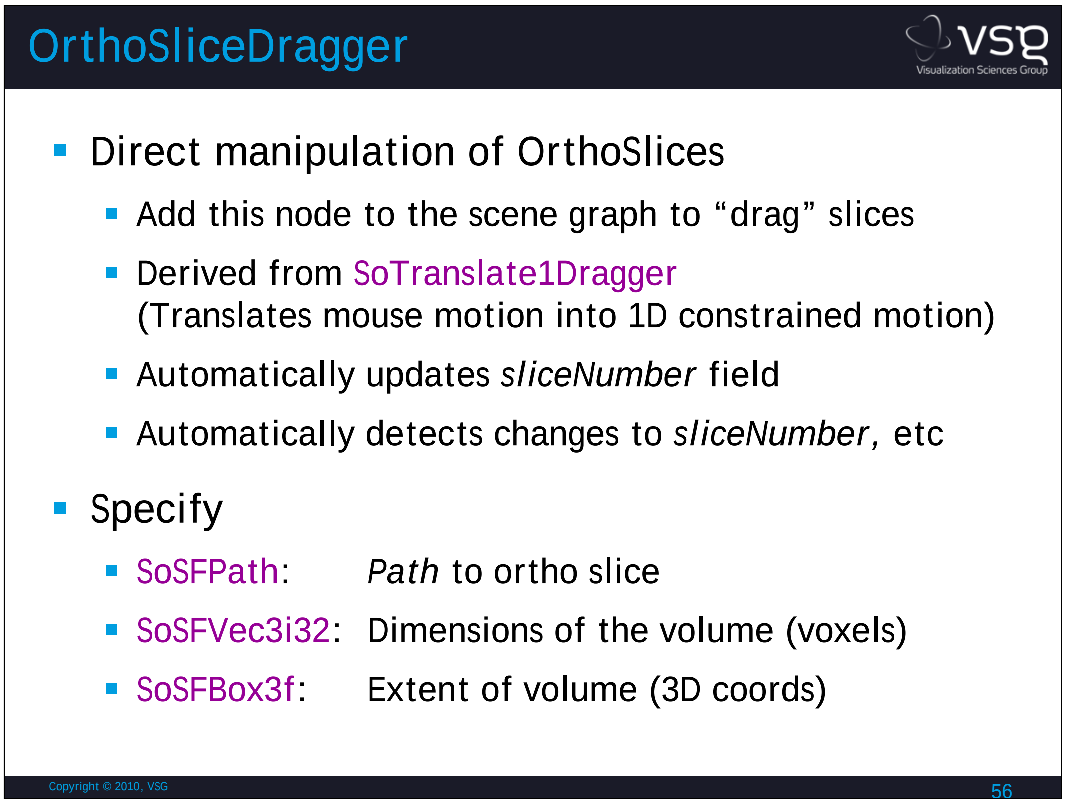
Two orthoslices



Now we’ll create the actual rendering nodes, in this case it’s two axis aligned slices, one each on the X and Z axes, positioned at the middle of the volume.

We’ll use our bounding box utility function to display the actual extent of the volume for reference and use the text labels to display the dimensions of the volume.

We could make this program interactive by adding some user interface, e.g. a slider, that allows the user to control the position of each slice. When the slider is changed, get the new value and update the *sliceNumber* field. As usual, Open Inventor will automatically re-render the scene when a field changes.



OrthoSliceDragger

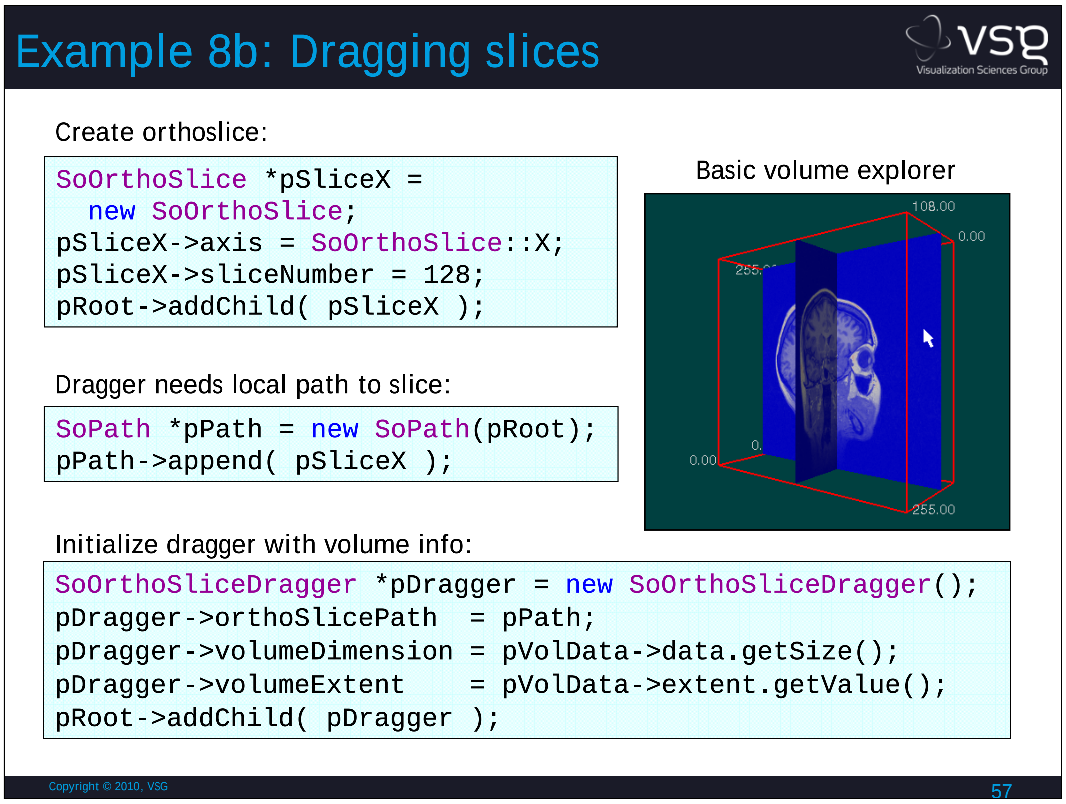
- Direct manipulation of OrthoSlices
  - Add this node to the scene graph to “drag” slices
  - Derived from SoTranslate1Dragger
    (Translates mouse motion into 1D constrained motion)
  - Automatically updates *sliceNumber* field
  - Automatically detects changes to *sliceNumber*, etc
- Specify

SoSFPath:	*Path* to ortho slice
SoSFVec3i32:	Dimensions of the volume (voxels)
SoSFBox3f:	Extent of volume (3D coords)

Directly dragging slices in the 3D window is such a common requirement that VolumeViz provides a node specifically for that purpose.

<u><CLICK!></u> It uses one of Open Inventor’s basic 3D input gadgets, called a dragger, to map 2D mouse inputs into intuitive 3D motion in the drawing window. Although all draggers have a default geometry that they can display, the OrthoSliceDragger does not display any geometry of its own. It detects mouse clicks on the OrthoSlice and automatically updates the *sliceNumber* field as the mouse is moved.

<u><CLICK!></u> It needs a *path* to the ortho slice to be controlled and also the dimensions and extent of the volume so it can constrain the motion to stay inside the volume. Next we’ll see how easy it is to add this complex interaction to our program.



Example 8b: Dragging slices

Create orthoslice:

```
SoOrthoSlice *pSliceX =
  new SoOrthoSlice;
pSliceX->axis = SoOrthoSlice::X;
pSliceX->sliceNumber = 128;
pRoot->addChild( pSliceX );
```

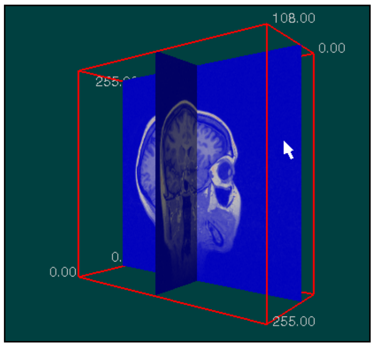
Dragger needs local path to slice:

```
SoPath *pPath = new SoPath(pRoot);
pPath->append( pSliceX );
```

Initialize dragger with volume info:

```
SoOrthoSliceDragger *pDragger = new SoOrthoSliceDragger();
pDragger->orthoSlicePath   = pPath;
pDragger->volumeDimension = pVolData->data.getSize();
pDragger->volumeExtent    = pVolData->extent.getValue();
pRoot->addChild( pDragger );
```

Basic volume explorer

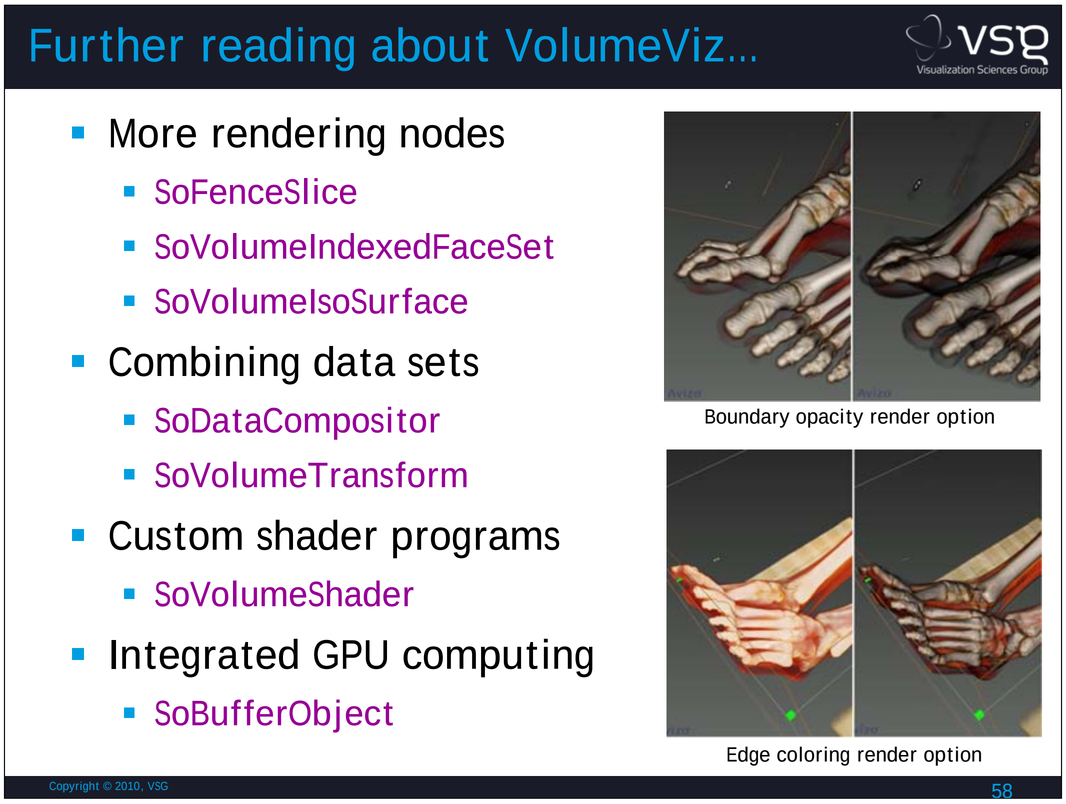


57

We'll modify the program so that when we create each slice, e.g. the X slice, we also create a dragger for that slice and add it to the scene graph.

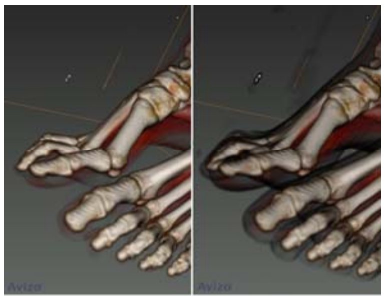
As a result, with no additional application code, the user can click on a slice and drag it along its axis. Of course the application will need to know when the slice has moved, or at least the new position. This is handled by setting callback functions on the slice dragger.

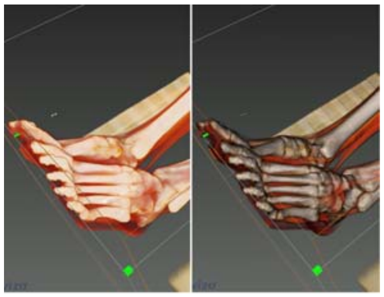
At this point we have already created a prototype/demo of an utility program that allows us to load a volume data set and explore by dragging slices through it.



Further reading about VolumeViz...

- More rendering nodes
  - SoFenceSlice
  - SoVolumeIndexedFaceSet
  - SoVolumeIsoSurface
- Combining data sets
  - SoDataCompositor
  - SoVolumeTransform
- Custom shader programs
  - SoVolumeShader
- Integrated GPU computing
  - SoBufferObject


Boundary opacity render option


Edge coloring render option

This short example just "scratches the surface" of the features provided by the shape classes. Here are just a few of the other things you may want to take advantage in your application. You can find more information about these features in the Open Inventor documentation.

Thank you for attending.

Questions?

www.vsg3d.com
www.openinventor.net

Feedback to:
mheck@vsg3d.com