Hands On with Open/Free VR Packages:
OpenSG & VRJuggler

www.opensg.org
www.vrjuggler.org

Dirk Reiners
dirk@louisiana.edu

Carsten Neumann
carsten@louisiana.edu

Dio Gonzalez
dioselin@louisiana.edu

Prerequisites

This tutorial assumes that you have

- basic C++ knowledge
  - Including basic STL use (vector)
- basic Computer Graphics knowledge
  - Polygon, Normal, Texture Coordinate and specular highlight should tell you something
- Downloaded, installed and tested the basic software package from http://external.lite3d.com/VR09Tutorial/
- Are willing and capable of downloading the addon package from there now and extract it into tutorial/tutorial

Compiling and running the Examples

- You need a working compiler
  - Windows: VS 2005 or higher
  - Linux: g++ 4.2 or higher
- Add tutorial/lib to your PATH environment
  - Start->System->Advanced->Environment Variables->PATH and append ; and the path to your unpacked tutorials
  - Create a new one called OSG_LOAD_LIBS with “OSGFileIO;OSGImageFileIO”
- You need cmake (2.6 or higher)
  - Point it at the source code (opensg_tut, vrjuggler_tut) and create a new build directory

Compiling and running the Examples

- Cmake…
  - Run Configure
    - Might need to add include dir for boost and GLUT in Advanced View
  - Run Generate, pick your compiler
    - Creates VS Project file
  - Compile Project and Run it.
- We are using a current version of OpenSG 2
  - Not released yet, documentation a little sparse

VRJuggler vs. OpenSG

- Why use both, what does VRJuggler have that OpenSG doesn't (and vice versa)?
- VRJuggler does everything until there is an open window to draw into
  - Motivation: bring existing OpenGL apps into VR
- OpenSG starts there
  - Do rendering, not device/UI management
- Overlap:
  - Both can do stereo/head-tracked rendering
    - But OpenSG doesn't handle the devices
  - Both can do clustering (in different ways)

Motivation

- Why use lower-level/programming libraries like OpenSG & VRJuggler?
  - Programming libraries vs. High-level system
- Pro:
  - More control / fewer constraints
  - Better Performance
- Con:
  - Higher initial resistance
  - More potential for errors
- How low do you want to go?
  - VRJuggler is happy with bare OpenGL

Motivation Scene Graphs

- OpenGL is very low-level
  - Immediate mode based
  - No object concept beyond display lists
  - User has full control...
    - ...and lots of ways to shoot himself in the foot.
    - Hardware can be picky about details
  - User has full responsibility
    - Needs to specify all data for every frame

Motivation Scene Graphs

- Scene Graphs are at a higher level
  - Retained mode
    - Build the data structures once
    - Use them as often as needed
  - Object-oriented
  - Knows the whole scene, can optimize better
  - Provide a lot of utility functions
- Compared to languages, OpenGL is like a macro assembler, while scene graphs are like C++.

When to use a Scenegraph

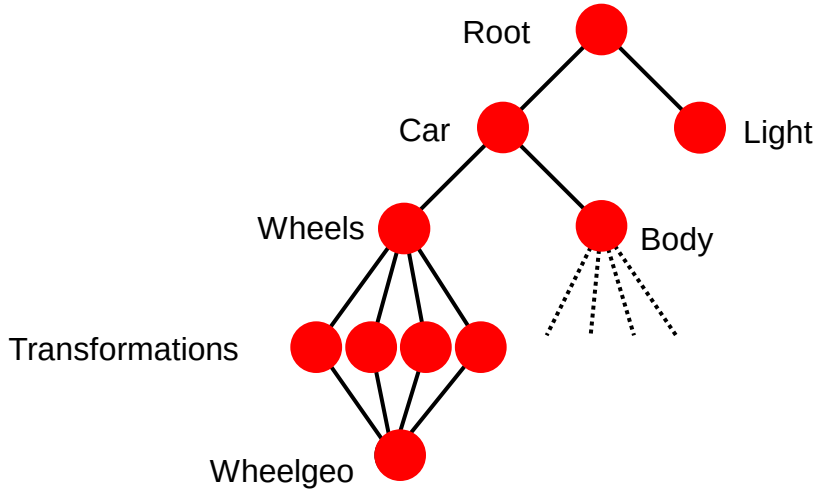
- Things scenegraphs do better than straight OpenGL:
  - Complex scenes: many different materials, large scenes where only a part is visible, complex hierarchy / transformations
  - Relatively static geometry
  - Scenes that are loaded from files
  - Specific features (particles, cluster, etc.)

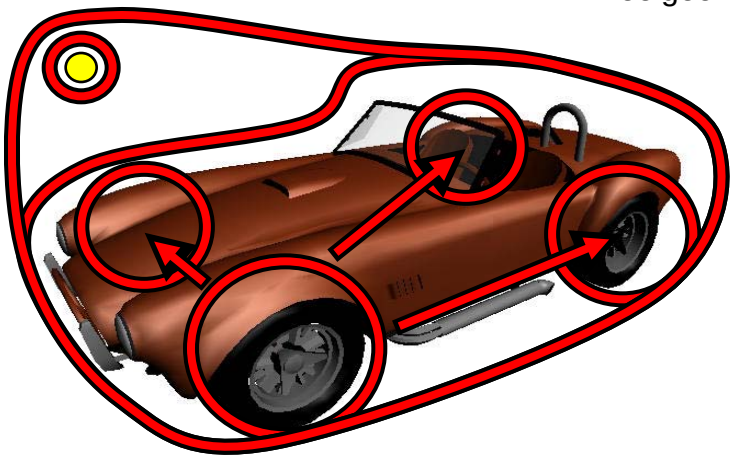
- Things that scenegraphs don’t do better than straight OpenGL:
  - Simple scenes (one object in the middle of the screen, a few simple additions)
  - Highly dynamic geometry

O.I.N.O.S.

- OpenSG is not Open Scene Graph!
- Two totally independent projects, with pretty much no overlap in people
  - But quite a bit in content, they're both modern scenegraphs after all
- Started at about the same time
- The two big Open Source scenegraphs
  - There are a bunch of smaller ones
  - And a ton of game engines…

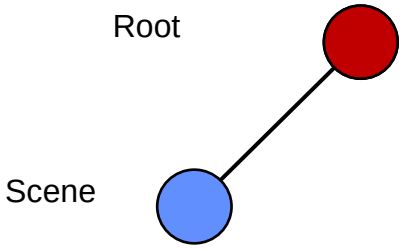
Scene Graph Structure





- Directed
- Acyclic
- Heterogenenous
- Logical Structure and Control in interior nodes
- Geometry / renderable Primitives in the leaves
  - E.g. polygonal meshes, particles etc.
- A lot of the flexibility of a scenegraph lies in the different node types it supports

Building a Trivial Graph




```
NodeRefPtr rootN = makeCoredNode<Group>();
NodeRefPtr sceneN = makeTorus(.5,2.,16,16);
rootN->addChild(sceneN);
commitChanges();
```

Questions…

- Why a RefPtr instead of just a Node*?
- What is a CoredNode?
- What is a Group?
- Where does the torus come from?
- What does commitChanges mean?

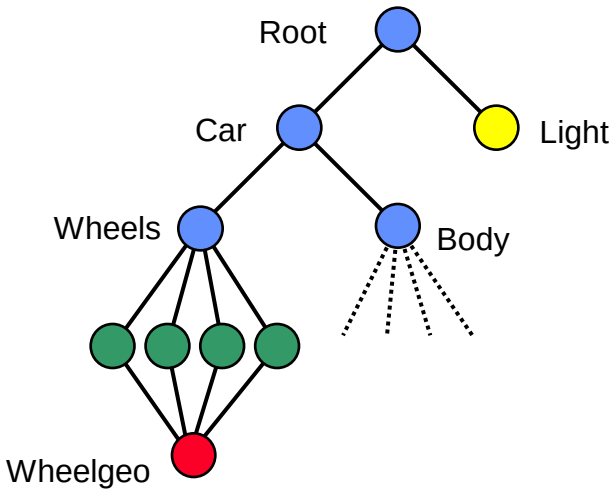
RefPtr: Memory Management or Who's going to clean up the mess?

- Keeping track of pointers in scenegraphs can get non-trivial
  - Many pointers to one node possible
    - Both inside the graph and outside
- Need robust method to keep track and clean up unneeded parts
- Solution: Reference Counting
  - Count the number of pointers to an object
  - When it falls to 0: delete the object
  - Hide bookkeeping in new type: RefPtr
    - Looks, acts just like a *, converts to it
    - Use always except for temporary variables

Reference Counting and Pointer Types

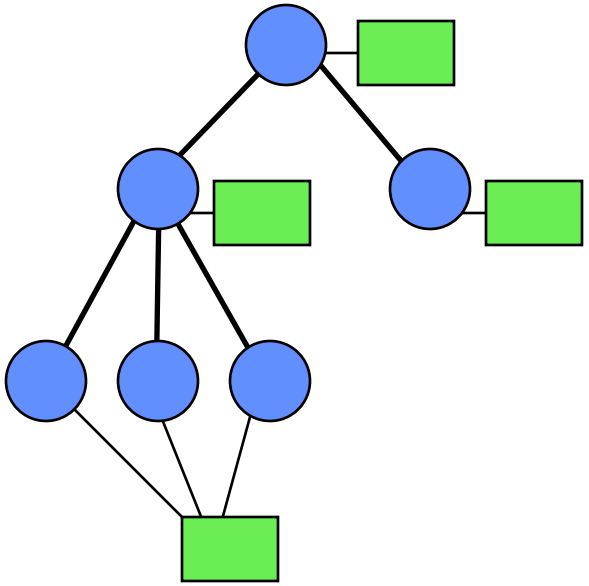
- New objects are created with a 0 refcount and returned as a RefPtr
  - Not created via `new`, have to use `TYPE::create()` factory method
  - Can't assign to a *, need to store them in a RefPtr or they will be deleted immediately
- Special Return Type: TransitPtr
  - RefPtr without conversion
  - Only useful for generator functions/methods
  - Don't worry about them, but if you see errors about TransitPtr: that's what it is
- There are other pointer types, more later

Nodes and Cores



- Sharing in the graph is important
  - Want to use same geometry multiple times
  - Want to use the same switch to affect multiple places, …
- How to do?
  - Attach a node to multiple parents
  - Each different parent just points to the same child
- Problems:
  - pointers / names can't be used to identify nodes
  - storage of position dependent data (e.g. accumulated matrix or world bbox) is impossible

Nodes and Cores



- Multiple parents not needed for data reuse
  - lean backbone (type **`Node`**)
  - reusable cores (type **`NodeCore`**)
    - define the actual behaviour
    - store the data
    - shareable
- Flipside: need to create two C++ objects for each object in the graph
  - Convenience function `makeCoredNode<TYPE>()`
  - Creates a NodeCore of type TYPE, a Node and connects the two
  - Problem: don't have the core to set parameters

The Complete Way

```
SwitchRefPtr flipper = Switch::create();
flipper->setChoice(Switch::ALL);
NodeRefPtr flipperN = Node::create();
flipperN->setCore(flipper);
commitChanges();
```

Slightly more convenient:

```
SwitchRefPtr flipper = Switch::create();
flipper->setChoice(Switch::ALL);
NodeRefPtr flipperN = makeNodeFor(flipper);
commitChanges();
```

Consequence: Sharing is Easy

- NodeCores can be shared between Nodes:

```
SwitchRefPtr flipper = Switch::create();
flipper->setChoice(Switch::ALL);
NodeRefPtr flipper1N = Node::create();
flipper1N->setCore(flipper);
NodeRefPtr flipper2N = Node::create();
flipper2N->setCore(flipper);
commitChanges();
```

- Changes to `flipper` will affect both nodes in the scenegraph

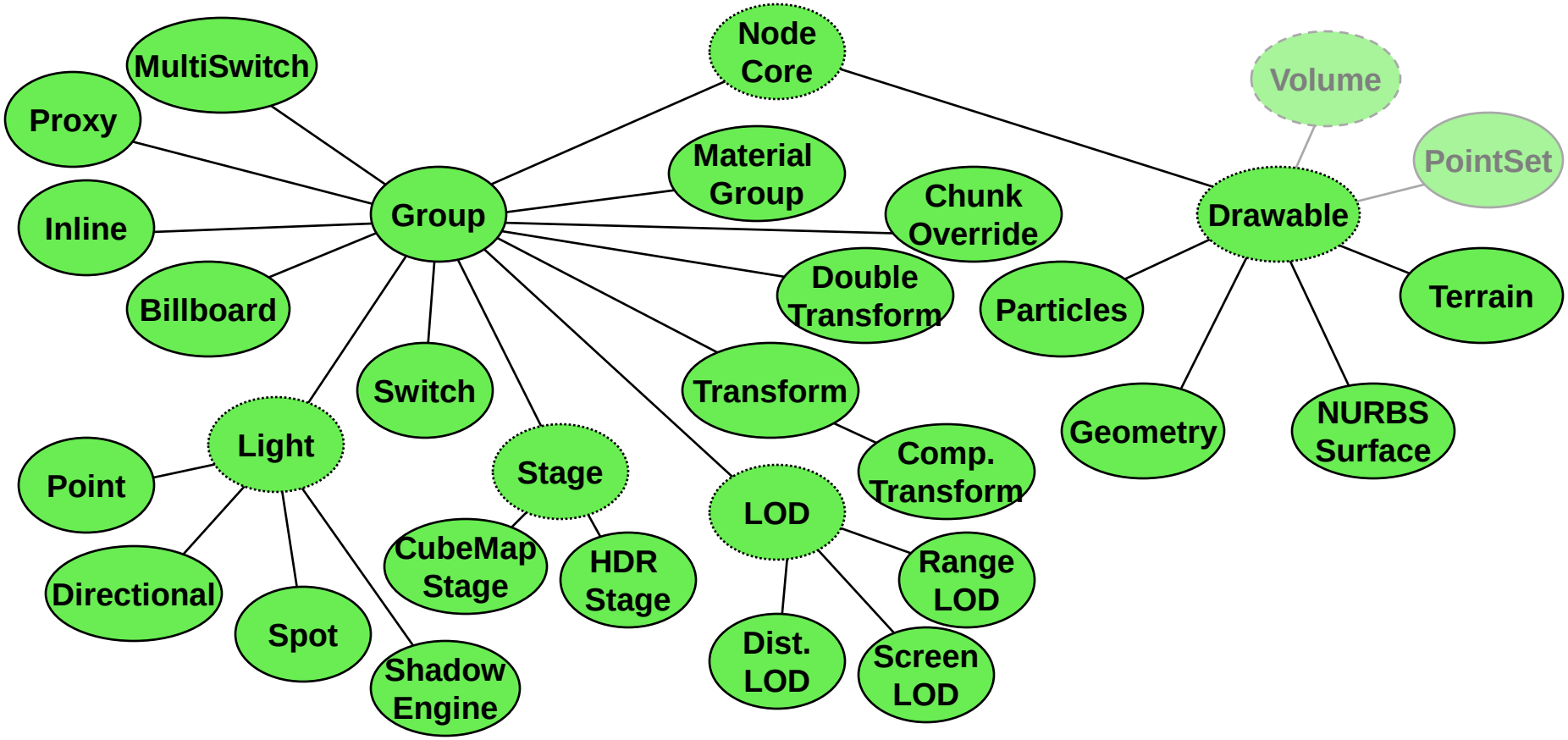
Sideline: Attachments

- Nodes (and some other objects) can store additional data: Attachments
  - Prime example: Name
- Stored as a map indexed by Attachment type and index
  - User-extendable
- Convenience functions for name:
  - setName(object, name);
  - getName(object);

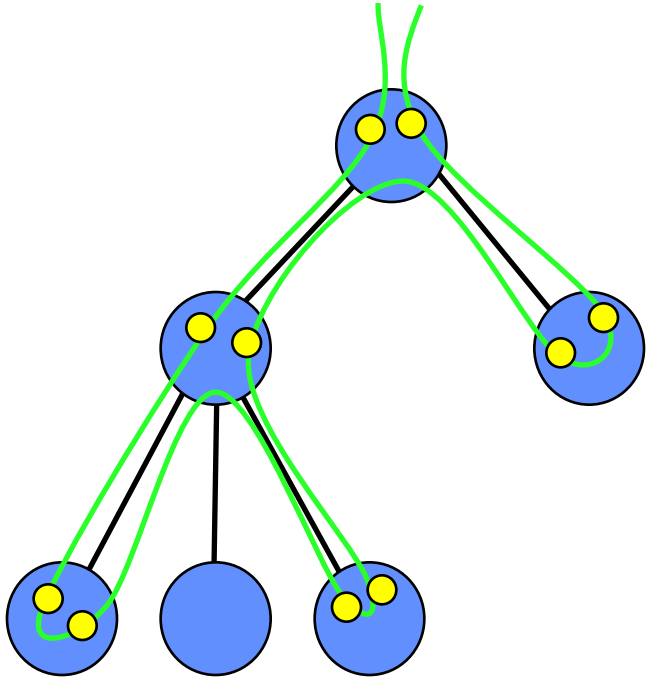
What's a Group? Or Node Types

- Major flexibility point for scenegraphs: different types of Nodes (or NodeCores in OpenSG)
- For many different functions:
  - Grouping logical/structural subgraphs (Composite pattern)
  - For defining higher-level rendering functions (multi-pass/stage methods)
  - Drawing only parts of the scene depending on different conditions
  - Changing how lower parts are drawn
    - Lighting
    - Materials
    - Transformations

OpenSG NodeCore Types (Partial)



How do Cores do it? Or: What do you do with the tree?



- Operation style: tree traversal
  - Start at a node, operate on it
  - Recurse through all, or some, children (if any)
  - Cleanup operation and return
- 3 control points:
  - Starting operation
  - Picking children to recurse through
  - Cleanup operation
- Many different levels of complexity…

Traversals

- Number of built-in traversals
  - Rendering
  - File writing
  - Intersection
  - Optimization
    - Striping
    - Find sharable parts and share them
    - Remove unneeded nodes
  - …
- Two different implementations: Actions & GraphOps

Actions / GraphOps / Direct Traversal

- Actions
  - Easy to add NodeCore types at runtime
  - A little more complicated to write
  - Built-in: Render, Intersect
- GraphOps
  - Can only deal with known NodeCore types
  - Easier to write
  - Number of builtins
- Direct Traversal
  - Either manually or using `traverse` helper function

Action: Using IntersectAction Example

- Shoot ray into scene, find first intersection

```
IntersectAction *act = IntersectAction::create();
act->setLine(Line(Pnt3f(0,0,0), Vec3f(1,0,0)));
act->apply(rootN);
if (act->didHit())
  std::cerr          << "Hit object “   << act-
 >getHitObject()
         << " tri "     << act->getHitTriangle()
           << " at "       << act->getHitPoint();
else
  std::cerr << "Nothing hit.“ << std::endl;
commitChanges();
delete act;
```

GraphOps: SharePtr Example

```
SharePtrGraphOp *op = SharePtrGraphOp();
op->setIncludes(“Material,StateChunk”);
op->traverse(rootN);
```

- GraphOps can be created and parameterized from strings
- Multiple GraphOps can be sequenced in a GraphOpSeq

```
GraphOpSeq seq(“Stripe() SharePtr(includes=Material,StateChunk)");
seq->run(rootN);
```

Simple and Useful Nodes

- Group
  - Just for structuring the graph
  - No operation, recurse through all children
  - No parameters
- Switch
  - For drawing subparts of the graph
  - Recurse through one or all of its children
  - Parameter: UInt32 choice
    - Index of child to recurse, or `Switch::ALL`

Simple and Useful Nodes

- MultiSwitch
  - Recurse through some of its children
  - Parameter: `UInt32 mode, UInt32 choices`
    - `mode`: `MultiSwitch`::`ALL`, `NONE` or `CHOICES`
    - `choices`: list of children to recurse through

- For switching large parts of the scene (i.e. many different nodes all throughout the graph) switches get unwieldy

Masked Traversal

- Often parts of the scene have different purposes
  - Collision vs. Rendering, Left vs. Right Eye, …
- Need to designate individual nodes to be one, the other, or both
  - And possibly for multiple purposes
- Solution: Traversal Mask
  - 32 bit mask in Node and Action
    - Action: copied from Viewport
  - Node is only traversed if bitwise and is != 0
- Allows arbitrary grouping for up to 32 groups

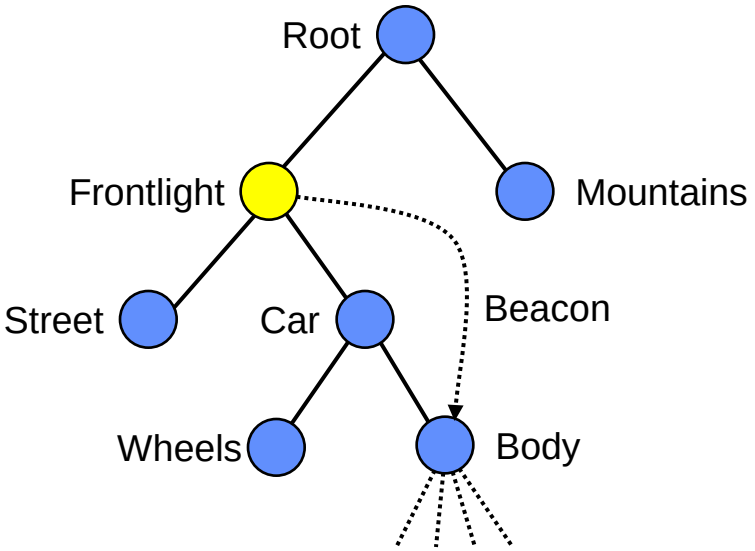
Transformations

- Static scenes can be nice, but moving things around is much more interesting
- Need to add transformation to the system
  - Transform NodeCore
    - Define Transformation as matrix
  - ComponentTransform
    - Define as product of translation/rotation/scale factors
- Transformations are valid for all cildren of the Node
- Transformations accumulate

Matrices

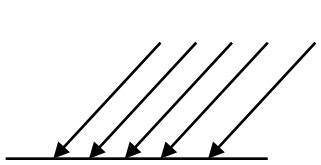
- (4x4) Matrix at the core
- OpenSG provides the usual set of matrix methods
  - Direct creation and from primitive transformations (translate, rotate, scale, …)
  - Multiplication of matrices and points/vectors
    - Vectors and Points are different, more later
  - Inversion, Transposition etc.

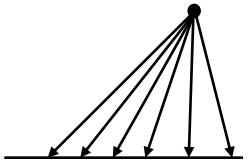
Lights

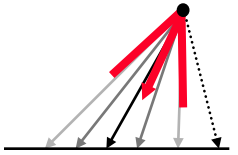


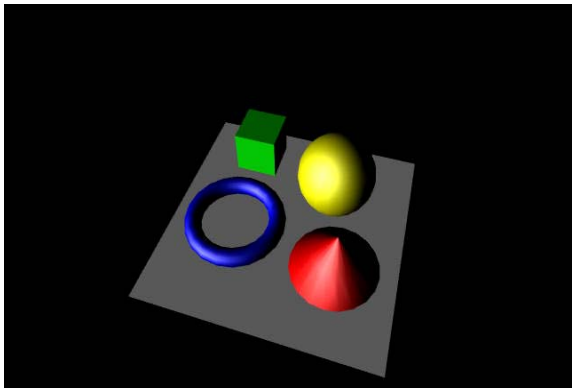
- Lights need two positions in the scenegraph:
  - One to define what is lit by them
    - Defined by the position in the tree, all children are lit
  - The second to define position/orientation
    - Defined by another node, the so called *beacon*
    - Cameras use beacons too. If a light uses the same beacon, it is head-mounted
- OpenSG doesn‘t have default lights, but the SimpleSceneManager helper adds a headlight

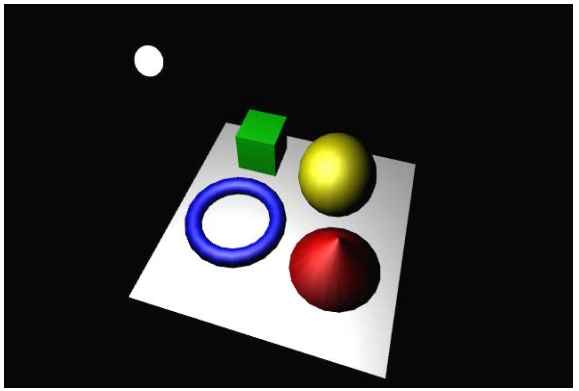
Light Types

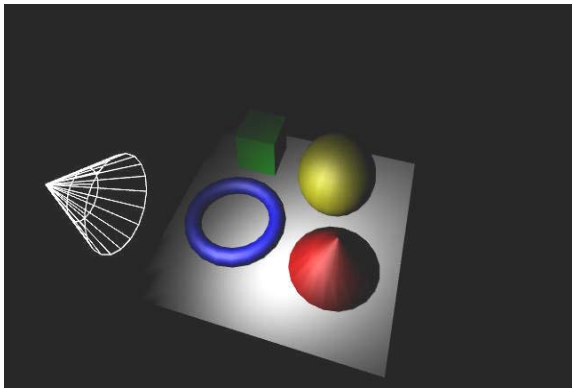












Directional

Point

Spot

Geometry

- Most common leaf node type
- Closely modeled after OpenGL
  - Mainly a set of arrays with uniform data types
    - Called Properties
  - A little more so than OpenGL
- All OpenGL primitives
  - …and all mixed in one geometry
- All OpenGL types
- Non-indexed, single-indexed and multi-indexed
- Only one material per Geometry though

Properties

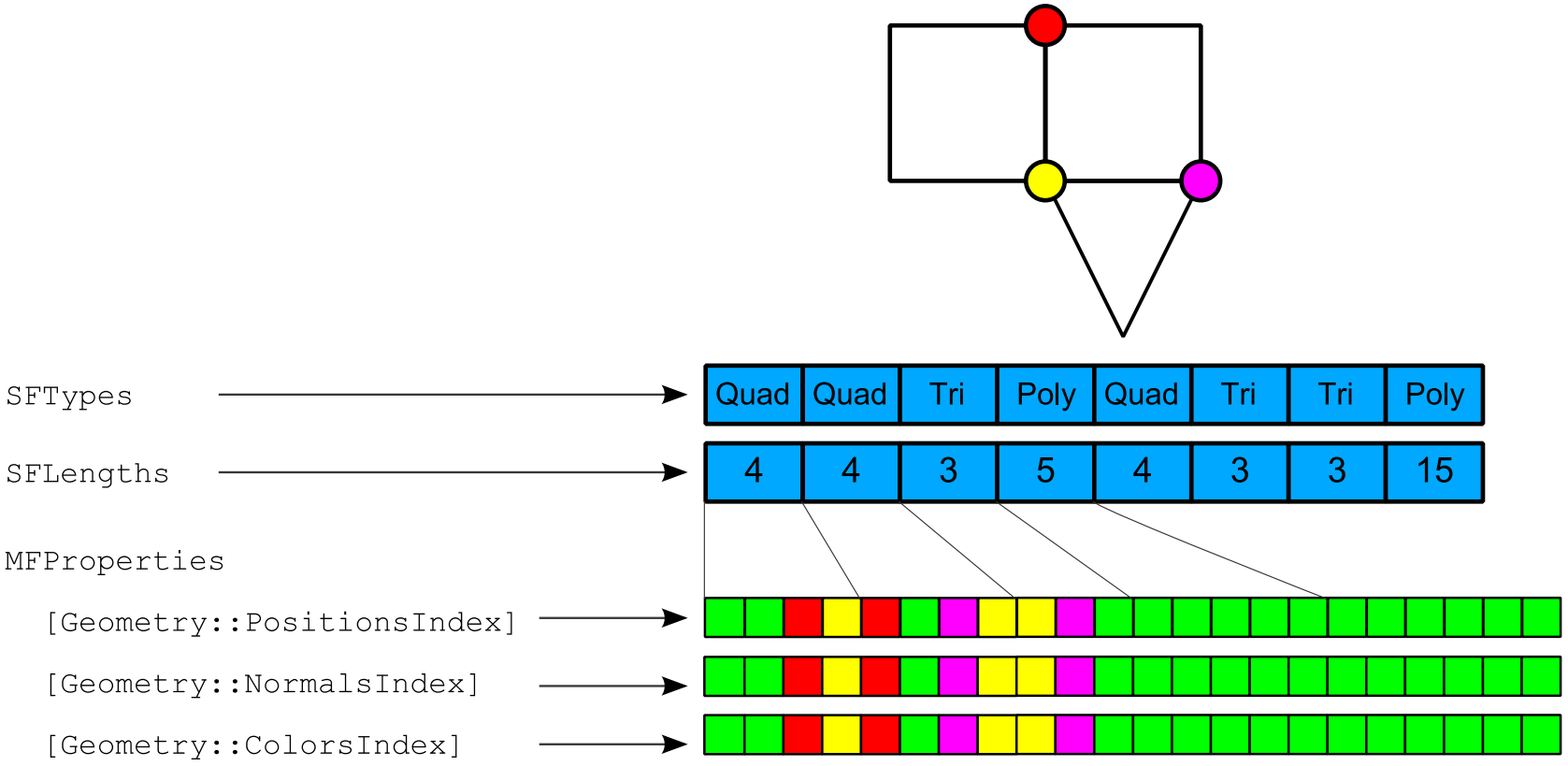
- Primarily just an array of uniform types
- Supports all OpenGL types
  - 1,2,3,4 D vectors and points
  - UInt8, UInt16, UInt32, Real32, Real64, …
- Naming scheme: Geo{TYPE}Property
  - E.g. GeoUInt8Property, GeoPnt3fProperty
- Simple interfaces
  - STL vector (push_back etc.)
  - Automatically converts type to internally stored one

Vectors / Points

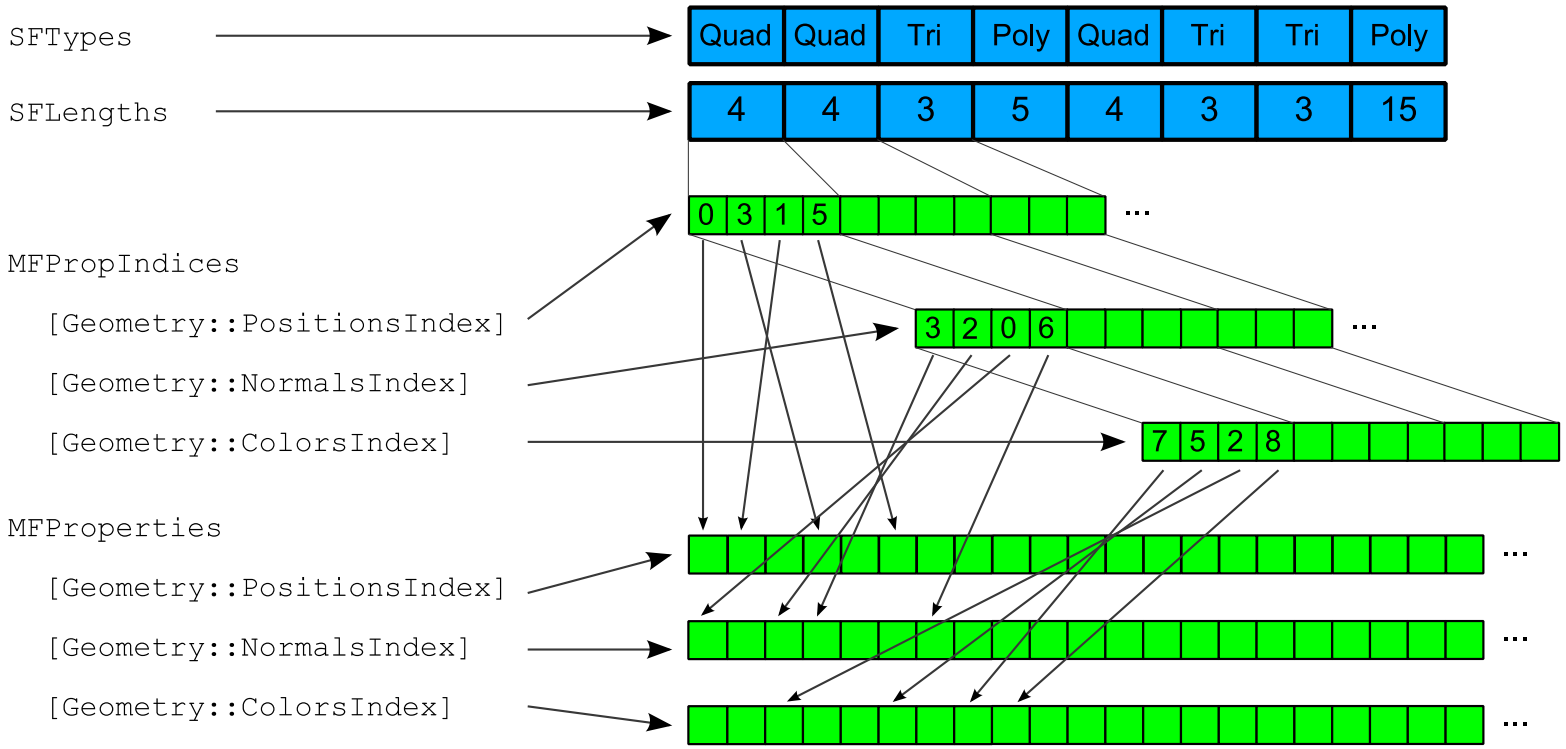
- Typed Variants
  - 1,2,3,4 dim, all types, e.g. Vec2f, Vec4ub, Pnt3f, Pnt4d, ...
- Standard value access interface []
- Difference Point / Vector
  - Points are positions in space
    - Operations : scale, negate, test, add vector, sub point, …
  - Vectors are directions in space (derived from point)
    - Operations : dot, cross, +, -, …

v = p – p **p = p + v** **v = v + v** **v = v – v** **p = p + p** 

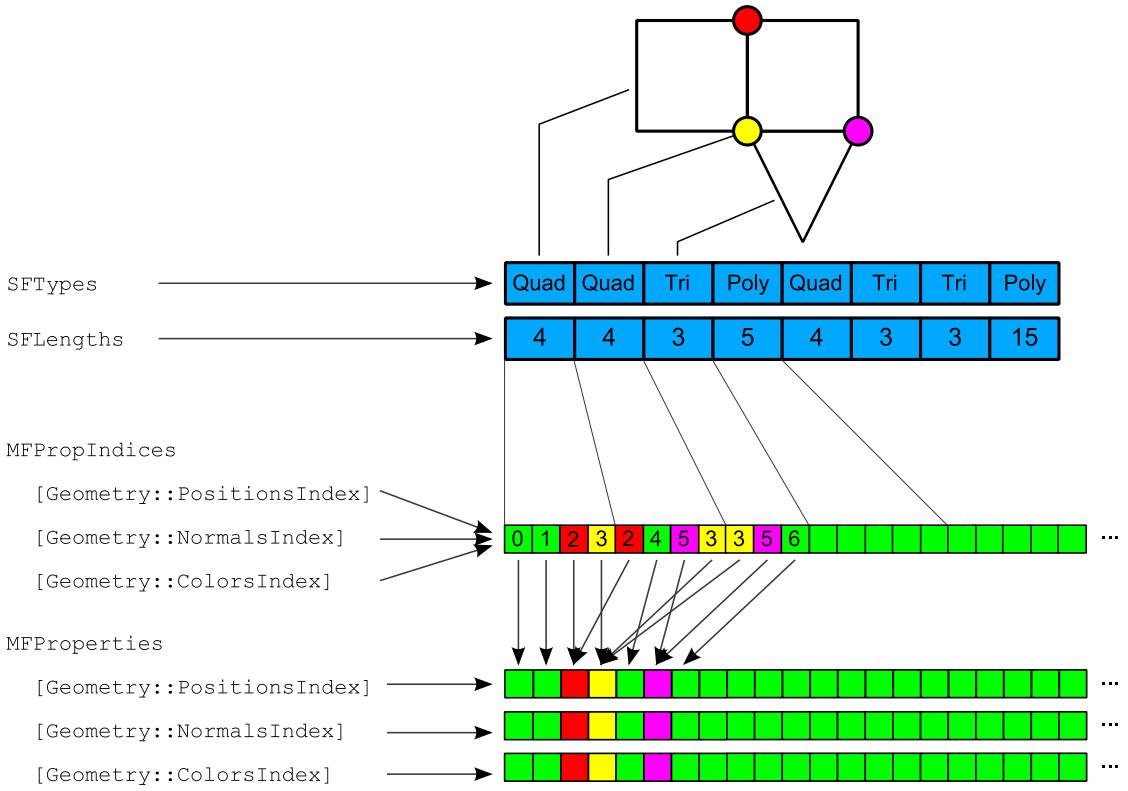
Non-Indexed Geometry



Multi-Indexed Geometry



Single-Indexed Geometry



Best Performance, so this is recommended to use!

Example: Creating a simple geometry

```
GeoUInt8PropertyRefPtr type =
  GeoUInt8Property::create();
type->addValue(GL_TRIANGLES);

GeoUInt32PropertyRefPtr lens =
  GeoUInt32Property::create();
lens->addValue(3);

GeoPnt3fPropertyRefPtr pnts =
  GeoPnt3fProperty::create();
pnts->addValue(Pnt3f(-1, -1, -1));
pnts->addValue(Pnt3f(-1, -1,  1));
pnts->addValue(Pnt3f( 1, -1,  1));

GeoVec3fPropertyRefPtr colors =
  GeoVec3fProperty::create();
colors->addValue(Color3f(1, 1, 0));
colors->addValue(Color3f(1, 0, 0));
colors->addValue(Color3f(1, 0, 1));
```

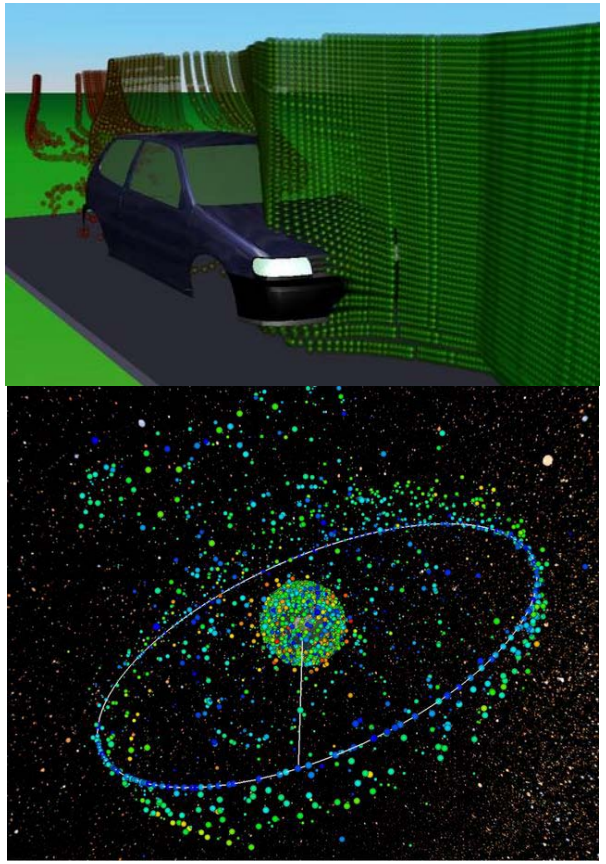
```
GeoUInt32PropertyRefPtr indices =
GeoUInt32Property::create();
indices->addValue(2);
indices->addValue(0);
indices->addValue(1);

GeometryRefPtr geo = Geometry::create();
geo->setTypes    (type);
geo->setLengths  (lens);
geo->setIndices  (indices);
geo->setPositions(pnts);
geo->setColors   (colors);
geo->setMaterial (getDefaultMaterial());

// put the geometry core into a node
NodeRefPtr n = Node::create();
n->setCore(geo);

commitChanges();
```

Easier: GeoBuilder

- Helper class to incrementally create Geometry
  - Support indexed and non-indexed geometry and arbitrary attributes
- Trivial Example:

```
GeoBuilder b;
b.begin(GL_TRIANGLES);
b.fullVertex(Pnt3f(1,1,1), Pnt3f(0,1,0), Color3f(0,0,1));
b.fullVertex(Pnt3f(-1,1,1), Pnt3f(0,1,0), Color3f(1,0,1));
b.fullVertex(Pnt3f(-1,-1,1), Pnt3f(0,1,0), Color3f(0,1,1));
b.end();
GeometryRefPtr geo = b.getGeometry();
```

Particles



- Renders different kinds of simple geometry
  - Points, Viewer-aligned quads, Arrows, ...
- Per-object Data
  - drawStyle:
  - drawOrder: Any, BackToFront, FrontToBack
- Per-particle data
  - Geometry-style Properties
    - Positions, Normals, Colors
  - Sizes: size of particle
- Note: this is only the rendering part, there is no simulation built into OpenSG
  - We have a little framework, if you're interested
  - Not enough time in this course

Loading Models

- OpenSG can load a variety of model formats:
  - 3DS, Collada , WRL, OBJ, OFF, STL, OpenFlight (very partial)
  - Internal binary format (OSB) for fast loading
- Loading of the file is handled by the SceneFileHandler

```
NodeRefPtr scene =
  SceneFileHandler::the()-
  >read("tie.wrl");
```

CommitChanges?

- OpenSG records all changes to the scene graph
  - Needed for Multithreading and Clustering
- Also used to update dependent data
- commitChanges gives all changed Objects a chance to update what needs to be updated
  - Inefficient to call after every single change
- Rule of thumb: call after finishing all changes for an object (or set of objects)
  - Need to be called before dependent data (e.g. bounding volume) can be read
- A few too many doesn't hurt, just don't do it after every single change

Multi-Threading

- Big Topic!
  - But not really introductory
    - Other things more important to get up and running
  - Talk to us afterwards if you're interested
- Just so much: OpenSG supports parallel changes to anything in the scenegraph (and most things outside of it)
  - One of the main things that make it unique
  - Very opaque for the user, hidden behind special pointer types, but that's it

Clustering

- More and more important (as seen in the instantReality part)
- iR's clustering is a pretty direct mapping into OpenSG
  - Classes are called pretty much the same
- Examples can use VRJuggler clustering
  - Different method, similar effect
- OpenSG can run tiled walls and performance clusters, but doesn't handle devices/tracking by itself
- Talk to us later if you're interested in details

OpenSG Materials

SimpleMaterial

- - Wraps OpenGL Material parameters:
  - Ambient, Diffuse and Specular color
  - Transparency
  - Lighting status (lit/unlit)
  - Interaction with lighting calculations (colorMaterial)
- Sufficient for untextured geometry

SimpleTexturedMaterial

- Like SimpleMaterial, with additional
  - Image
  - Mipmap filtering control (minFilter, magFilter)
  - Interaction of texture color with material color (envMode, e.g. GL_REPLACE, GL_MODULATE)
- Convenient for quickly adding a single texture to an object

Connecting Materials with Drawables

- MaterialDrawable is the base of everything that has a Material, in particular Geometry

```
geo->setMaterial(myMat);
```

- MaterialGroup, determines Material for a subtree, taking precedence over what is already there

```
matGroup->setMaterial(myMat);
```

- More fine grained control is possible with StateChunks

StateChunk – Basic unit of State

- All OpenGL State is stored in StateChunks
- Container for closely related settings that often change together
- Defaults are the same as OpenGL
- Combination of multiple StateChunks determines appearance of rendered objects
- Can (and should!) be used in multiple materials

BlendChunk

- Alpha blending (srcFactor, destFactor)

```
blendChunk->setSrcFactor(GL_SRC_ALPHA);
blendChunk->setDestFactor(GL_ONE_MINUS_SRC_ALPHA);
```

- Alpha testing (alphaFunc, alphaValue)

```
blendChunk->setAlphaFunc(GL_GREATER);
blendChunk->setAlphaValue(0.5f);
```

- Marks the Material as transparent
- Transparent objects are rendered after opaque ones for correct blending

PolygonChunk

- Backface culling (cullFace)

```
polyChunk->setCullFace(GL_BACK);
```

- Orientation – which side is front (frontFace)

```
polyChunk->setFrontFace(GL_CCW);
```

- Filled/Wireframe/Point mode for front and back face (frontMode, backMode)

```
polyChunk->setFrontMode(GL_LINE);
polyChunk->setBackMode(GL_FILL);
```

- Polygon offset

Other Chunks

- MaterialChunk
  - Same setting as SimpleMaterial
- TwoSidedLightingChunk
  - Enable lighting calculations for front and back face polygons
- ShadeModelChunk
  - Choose between flat and smooth shading

ChunkMaterial – Grouping Chunks

- Container for multiple StateChunks
- Some types of chunks can be present more than once (e.g. textures, clip planes)
- Each chunk type has one or more slots that can be filled
  - Used to assign e.g. textures to texture units
- Base of SimpleMaterial
  - Missing settings can be added by adding the relevant chunk

ChunkOverrideGroup

- Overrides/Adds chunks to Materials
  - Allows to quickly change appearance for whole subtrees
- Accumulated during tree traversal
  - Same chunks can be overridden multiple times, closest override wins
  - Especially useful with shader (GLSL) related chunks

Example – Drawing a subtree as wireframe

```
PolygonChunkRefPtr polyChunk = PolygonChunk::create();
polyChunk->setFrontMode(GL_LINE);
polyChunk->setBackMode(GL_LINE);

ChunkOverrideGroupRefPtr cog =
 ChunkOverrideGroup::create();
cog->addChunk(polyChunk);

NodeRefPtr cogN = makeNodeFor(cog);

// add cogN on top of a subtree
```

SortKey – Influencing Render Order

- Materials with smaller sort key values are rendered first
- Sometimes can be used to manually correct alpha blending problems
- Chunks present on a material determine default
- Ensures that Materials with similar chunks are rendered together – reduces state changes

Textures

- Two chunks
  - TextureObjChunk – The texture object itself
  - TextureEnvChunk – How it is applied
- Pixel data is stored in an Image
- No further distinction between 1D, 2D, 3D, Cube textures

Image

- One type for everything that has pixels
  - Mipmaps
  - Multiple Frames (animated Images)
  - Multiple Slices (3D Images)
  - Compressed Texture data
- Loading from files

```
ImageRefPtr img = ImageFileHandler::the()->read(“fileName”);
```

- Supported formats
  JPG, PNG, TGA, TIF, GIF, DDS, HDR, MNG, MTD, …

Image II

- Variety of data formats
  - RGB(A)
  - BGR(A)
  - Luminance
  - DXT1 – DXT5
  - Signed integer variants
- … and types
  - 8, 16, 32 bit per channel integer
  - 16, 32 bit per channel floating point

Textures – TextureObjChunk

- Represents a single texture map (1D, 2D, 3D, Cube)

```
texObj->setImage(img);
```

- Filter controls (minFilter, magFilter)

```
texObj->setMinFilter(GL_LINEAR_MIPMAP_LINEAR);
texObj->setMagFilter(GL_LINEAR                );
```

- Wrapping mode in S,T,R direction

- Can refer to an Image with NULL data for Textures that are only in GPU memory

Textures – TextureEnvChunk

- Interaction of texture colors with material colors (envMode)

```
texEnvChunk->setEnvMode(GL_MODULATE);
texEnvChunk->setEnvMode(GL_REPLACE);
```

- Gives access to older texture shader extensions (NV_texture_shader{,2,3})

Example – Adding a Texture to a Material

```
ImageRefPtr img =
  ImageFileHandler::the()->read(“carpet.jpg”);

TextureObjChunkRefPtr texObj = TextureObjChunk::create();
texObj->setImage(img);

TextureEnvChunkRefPtr texEnv = TextureEnvChunk::create();
texEnv->setEnvMode(GL_MODULATE);

chunkMat->addChunk(texObj);
chunkMat->addChunk(texEnv);
```

Shaders

- Written in GLSL or Cg (when supported by the driver)
- ShaderProgram – Independently compileable piece of shader code
- ShaderProgramChunk – Collects multiple ShaderPrograms
- ShaderProgramVariables – Stores (uniform) variables passed to the shader code
- ShaderProgramVariablesChunk – Collects multiple ShaderProgramVariables

Shaders – Creating

- Create a ShaderProgram of the desired type

```
ShaderProgramRefPtr vp = ShaderProgram::createVertexProgram();
ShaderProgramRefPtr gp = ShaderProgram::createGeometryProgram();
ShaderProgramRefPtr fp = ShaderProgram::createFragmentProgram();
```

- Set the code

```
vp->setProgram(std::string(“ … “));
```

- Or load it from a file

```
vp->readProgram(“fileName”);
```

Shaders – Adding uniform variables

- Directly to the ShaderProgram

```
vp->addUniformVariable(“sunPosition”, Pnt3f(100.f, 300.f, 50.f));
```

- Separate ShaderProgramVariablesChunk object

```
ShaderProgramVariablesChunkRefPtr varChunk =
  ShaderProgramVariablesChunk::create();
varChunk->addUniformVariable(
  “moonPosition”, Pnt3f(-200.f, -300.f, 50.f));
```

- Modify values with `updateUniformVariable`
- Remove variables with `subUniformVariable`

Shaders – System Variables

- Access to variables OpenSG tracks internally during the tree traversal

```
vp->addOSGVariable(“OSGWorldMatrix”);
vp->addOSGVariable(“OSGViewMatrix”);
vp->addOSGVariable(“OSGLight0Active”);
```

- In the shader code these are declared just like any other uniform

```
uniform mat4 OSGWorldMatrix;
uniform bool OSGLight0Active;
```

Shaders – Putting it all together

- Add ShaderPrograms to a ShaderProgramChunk

```
ShaderProgramChunkRefPtr spChunk = ShaderProgramChunk::create();
spChunk->addShader(vp);
spChunk->addShader(fp);
```

- … add the program chunk to a material

```
mat->addChunk(spChunk);
```

- … as well as the variables chunk (optional)

```
mat->addChunk(varChunk);
```

Example Outline – Per pixel lighting

- Create Vertex and Fragment ShaderPrograms and load the GLSL code from a file

```
ShaderProgramRefPtr vpPPL =
  ShaderProgram::createVertexProgram();
ShaderProgramRefPtr fpPPL =
  ShaderProgram::createFragmentProgram();
vpPPL->readProgram(“pixellight.vp.glsl”);
fpPPL->readProgram(“pixellight.fp.glsl”);
```

- Create a chunk to hold the two program objects

```
ShaderProgramChunkRefPtr shaderChunk =
  ShaderProgramChunk::create();
shaderChunk->addShader(vpPPL);
shaderChunk->addShader(fpPPL);
```

Example Outline – Per pixel lighting (Cont.)

- Create uniform variables to get access to the texture and light status

```
ShaderProgramVariablesChunkRefPtr varChunk =
  ShaderProgramVariablesChunk::create();
varChunk->addUniformVariable(“diffuseTexture”,  0);
varChunk->addOSGVariable    (“OSGLight0Active”   );
```

- Add both chunks to a Material

```
mat->addChunk(shaderChunk);
mat->addChunk(varChunk   );
```

Example – Per pixel lighting (Vertex Shader)

```
varying vec3 normalVec, lightVec;

void main()
{
    vec4 ecPos;
    normalVec  = normalize(gl_NormalMatrix * gl_Normal);
    ecPos      = gl_ModelViewMatrix * gl_Vertex;
    lightVec   = normalize(vec3(gl_LightSource[0].position – ecPos));
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position    = ftransform();
}
```

Example – Per pixel lighting (Fragment Shader)

```
varying vec3 normalVec, lightVec;

uniform bool OSGLight0Active;
sampler2D    diffuseTexture;

vec4 computeLight(int index, vec3 lightVec, vec4 diffCol);

void main(void)
{
    vec4 color = vec4(0., 0., 0., 0.);
    if(OSGLight0Active)
    {
        vec4 diffCol = texture2D(diffuseTexture, gl_TexCoord[0]);
        color += computeLight(0, lightVec, diffCol);
    }

    gl_FragColor = color;
}
```

Graphics Effects
and
Advanced Techniques

Stage – Redirecting rendering output

- Base of family of types
  - Render output control – FrameBufferObject
  - Pre/Post processing of generated image data
- Implement a specific effect
  - High Dynamic Range (HDR) rendering
  - Shadow Maps
  - Display Filter (Color, Geometry, Resolution correction)
- Building block for user created effects (SimpleStage)

SimpleStage – Common Fields

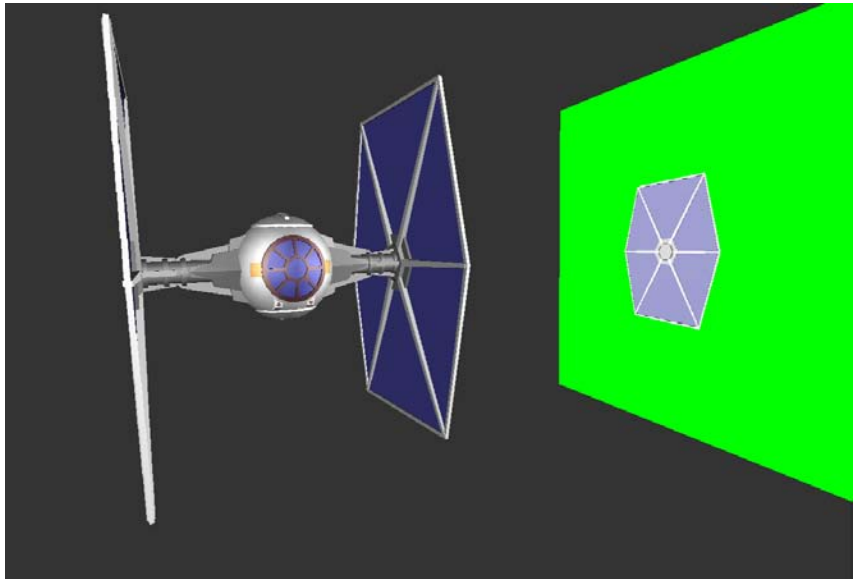
- Fields renderTarget and inheritedTarget determine where pixels are written to
- Camera to change point of view and projection
- Values for left, right, top, bottom to restrict rendering to a subrectangle of the renderTarget
- pre/postRenderCallbacks
  - List of callback functors
  - Typically used for image processing operations

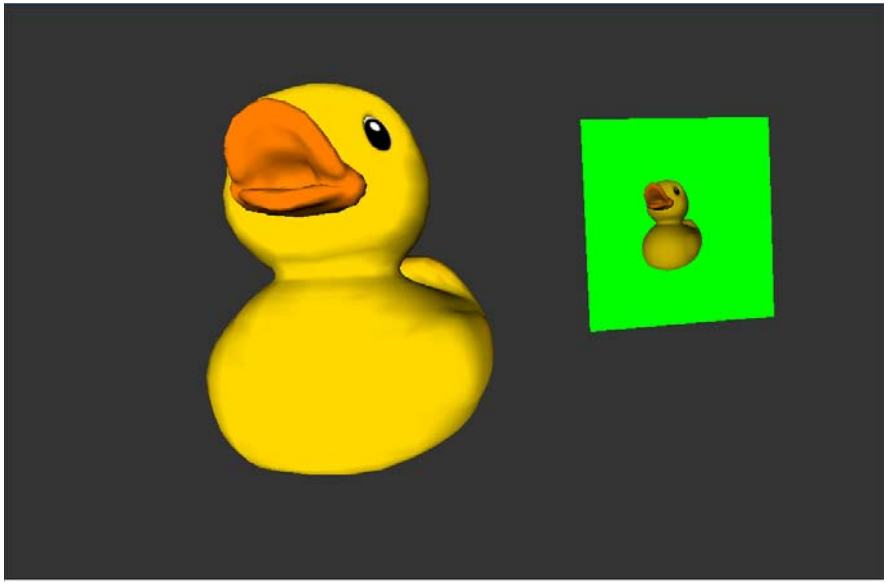
Example Outline – Building a Mirror

- Render the scene from a second point of view
- Use the image as a texture when rendering the regular view

- Create a SimpleStage
  - Add a FrameBufferObject as render target
    - Add a TextureBuffer as color attachment
    - Add a RenderBuffer as depth attachment
  - Add a new a Camera at the reflected viewer position
- Apply the generated texture (color attachment) to the mirror object
  - The TextureObjChunk of the TextureBuffer can be directly added to the mirrors material

Example Outline – Building a Mirror – Images





ShadowStage

- Implements 6 shadow map algorithms
  - Percentage closer filtering
  - Perspective shadow maps
  - Variance shadow maps
  - Percentage closer soft shadows
  - Dither shadow maps
  - Standard shadow maps
- Smoothness factor to control filtering quality vs. performance

ShadowStage – Adding shadows to a scene

- Choose an algorithm to use

```
shadowStage->setShadowMode(ShadowStage::PCF_SHADOW_MAP);
```

- Choose a map size

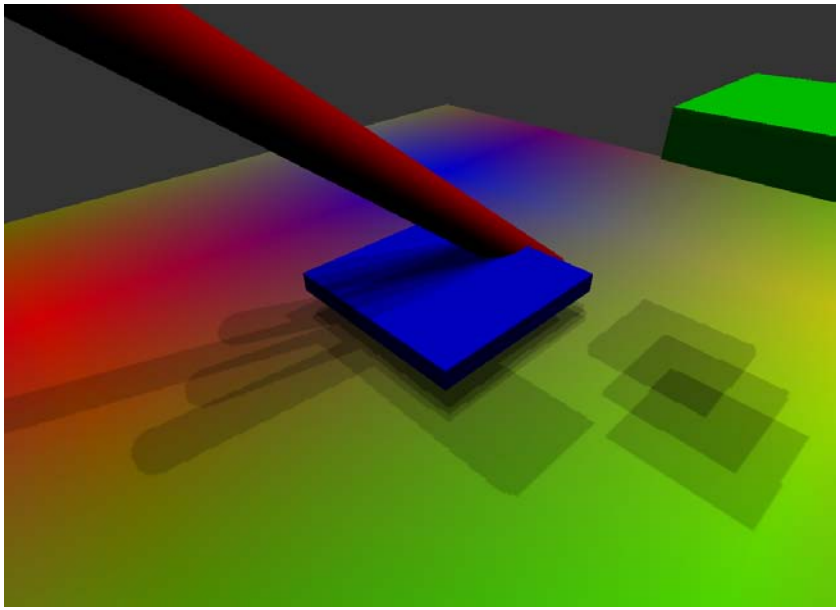
```
shadowStage->setMapSize(1024);
```

- Enable autoSearchForLights mode or add lights explicitly

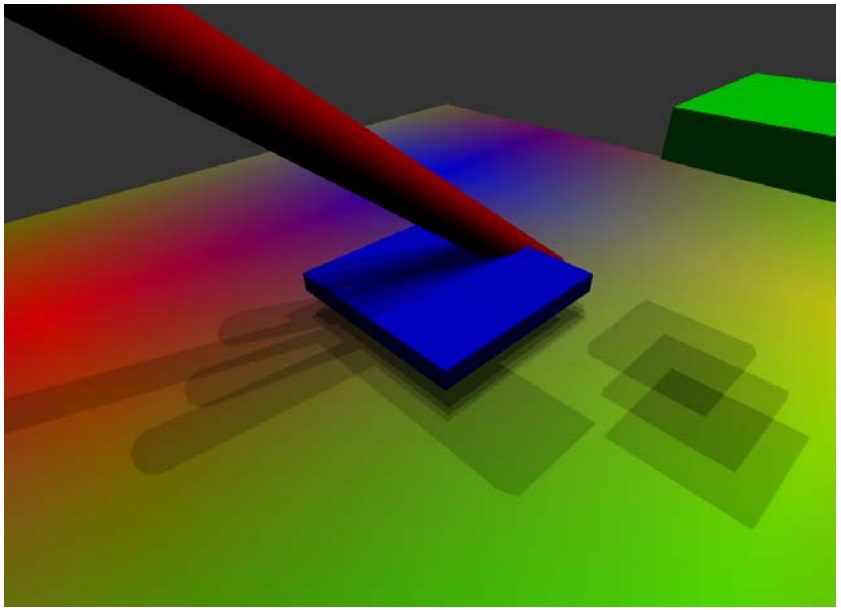
```
shadowStage->setAutoSearchForLights(true);
shadowStage->editMFLightNodes()->push_back(lightNode);
```

- Add on top of an existing scene

ShadowStage – Example Images

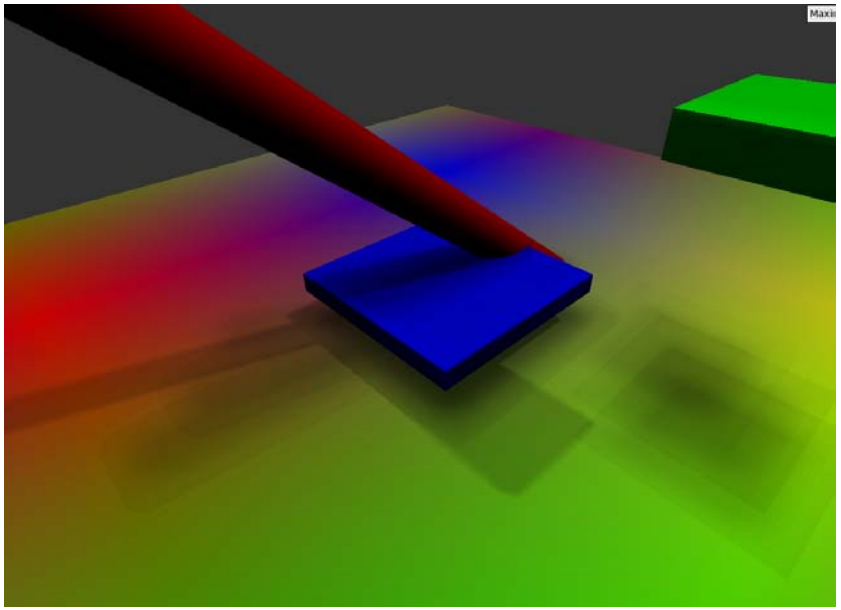


Standard

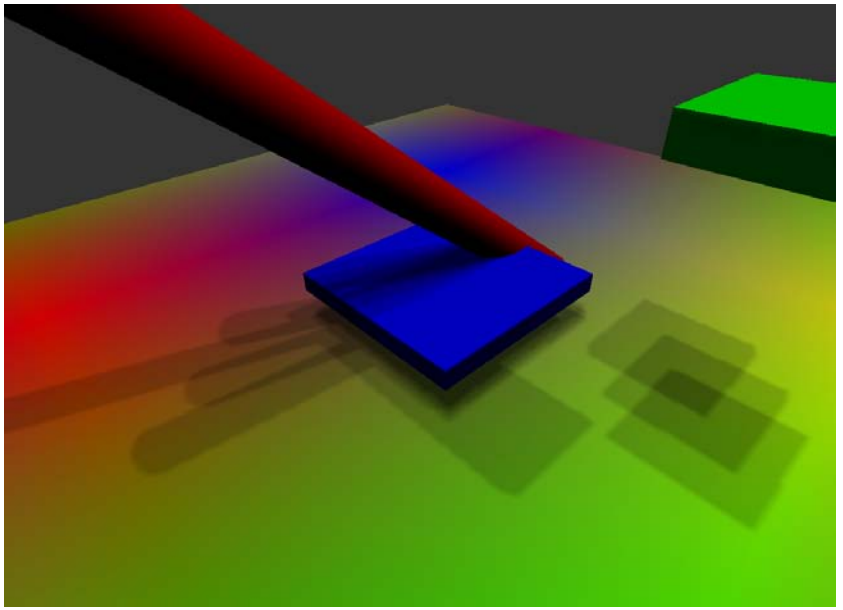


Perspective

ShadowStage – Example Images II



Percentage Closer Soft Shadows (PCSS)



Percentage Closer Filtering (PCF)

HDRStage

- Renders the scene to a floating point texture buffer
- Applies tone mapping and blur operations to the buffer
- Combines tone mapped and blurred buffers into final image

HDRStage – Setup

- Select blur kernel size

```
hdrStage->setBlurWidth(3.f);
```

- Select strength of blur and bloom effect

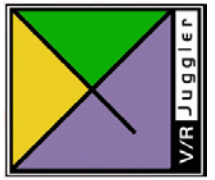
```
hdrStage->setBlurAmount(0.5f);
hdrStage->setEffectAmount(0.2f);
```

- Select exposure and gamma value

```
hdrStage->setExposure(8.f);
hdrStage->setGamma(0.5f);
```

- Add on top of scene

Up next: VRJuggler



Summary

- OpenSG and VRJuggler provide a wide array of tools to create interactive and immersive 3D applications
- It is more effort than just running an existing system
- But it gives more flexibility, especially for unconventional applications
  - If all you want to do is flying through a model it’s probably overkill
  - But if you need to write new and exciting ways of dealing with complex data, they are a good base.