Scene Representation
Our overall motivation...

- Use common rendering lib for many rendering applications

```
Rendering.dll
Abstract interfaces:
  - IRenderer
  - IResourceManager

RenderingGLES.dll
RenderingD3D.dll

Applications use
Abstract API

Concrete implementations

Easy, isn't it?
```
Many application areas...

Flooding simulation

Industrial CT Scan Visualization

Infrastructure Visualization/Planning: GearViewer

Geological annotations on surfaces
Versatile scene representation needed

- Each application has its own domain “language”
- Common task: describe static or dynamic 2D/3D scenes
- We look for a description language which works for a *wide range* of applications.
- How could we model different scenes?
  - Domain specific notation with built-in entities such as
    - Bridges, bones, buildings, dip-and-strike tools for geospatial applications etc.
    - This notation we will later call the *semantic scene graph*
  - Notation for 3D objects, interaction etc.
    - Talk about geometries, transformations
    - This notation we will later call the *rendering scene graph*
    - no domain logic
Scene description in OpenGL

- Example: old school OpenGL
- OpenGL had abstraction mechanisms built in:
  - Matrix stack
- Next level of abstraction:
  - Move OpenGL code into utility functions.
  - Additional utility functions can be used to modify graphics state.

```c
void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glTranslatef (-1.0, 0.0, 0.0);
    glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
    glTranslatef (1.0, 0.0, 0.0);
    glPushMatrix();
    glScalef (2.0, 0.4, 1.0);
    glutWireCube (1.0);
    glPopMatrix();
    glTranslatef (1.0, 0.0, 0.0);
    glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
    glTranslatef (1.0, 0.0, 0.0);
    glPushMatrix();
    glScalef (2.0, 0.4, 1.0);
    glutWireCube (1.0);
    glPopMatrix();
    glPopMatrix();
    glutSwapBuffers();
}
```

OpenGL Red Book 1.1, robot.c
Scene decomposition

```
void RenderRobotArm()
{
    RenderArm(lowerArmTransform);
    RenderArm(upperArmTransform);
    RenderHand(handTransform);
}

void RenderArm() {..}

void RenderHand()
{
    RenderFinger(trafo);
    ...
}
```
Attributes for scene graphs

Towards an explicit data representation of scenes

- Represent each geometric entity as node
- Special purpose nodes for changing appearance
  - Transformation nodes
  - Shader nodes
  - Material nodes
  - Specify light etc…
- Nodes can be **composed together** in order to make more powerful nodes
- Scene description can be modified by rendering application
  - e.g. nodes can be stored in variables, modified, used at various points etc.
The traditional rendering scene graph

- When using explicit data representation for entities and state-changing nodes, we arrive at a simple scene graph.
- The scene can be rendered by traversing the scene graph.

```
Graphics.setViewTrafo (...)
Graphics.setShader  (...)
Graphics.render     (...)
Graphics.setViewTrafo (...)
...
```

Traverse with sideeffects means: walk over structure and issue appropriate commands to the underlying graphics hardware.
Scene graphs ‘most general’ description

- Maya for example uses node based scene description: Hypergraph.
- Most other engines as well
  - Most scene exchange formats are some sort of scene graph
    - VRML, SVG, COLLADA, X3d,...
- A simple object list is a (rather boring) scene graph as well
- Techniques mentioned here carry over to other areas
  - Most UI frameworks use some sort of scene graph (e.g. WPF)
  - Modelling tools often use graph structure for defining materials etc.
Example: Maya Hypergraph

- Helps to structure scene
- Transformation hierarchy
Example: Renderman scene description language

- Hierarchical scene description
- Various attributes and node types
  - ConcatTransform
  - Transform
  - But also light properties
- ASCII description

https://community.renderman.pixar.com/article/400/cornell-box.html?l=r
Option "statistics" "endofframe" [1]
Exposure 1.0 1.0

#Display "cornell_box.exr" "openexr" "rgba"
Display "cornell_box.exr" "it" "rgba"

#Integrator "PxrPathTracer" "handle" "int numLightSamples" [4]
"int numBxdfSamples" [4]

Hider "raytrace"
  "constant string integrationmode" ["path"]
  "constant int incremental" [1]
  "int minsamples" [32]
  "int maxsamples" [1032]

Integrator "PxrVCM" "PxrVCM"
  "int maxPathLength" [10]
  "int mergePaths" [1]
  "int connectPaths" [1]

PixelVariance .007
Format 500 500 1.0
ShadingRate 1.0

Projection "perspective" "fov" [ 39.14625166082039 ] # lens 45.0, aspect 1.0
Rotate 180 0 1 0 # right handed
Scale -1 1 1 # right handed
Other boxes and light omitted.....

# cornell_box
Attribute "identifier" "name" "cornell_box"
AttributeBegin
ConcatTransform [
   -1.0 -1.5099580252808664e-07 0.0 0.0....
]
# cbox_green [2]
Opacity [1.0 1.0 1.0]
Color [0.0 0.5 0.0]
Bxdf "PxrLMDiffuse" "cbox_green2" "color frontColor" [0.0 0.5 0.0]
PointsPolygons
[ 4 ]
[ 0 1 2 3 ]
"P" [
   0.0 0.0 0.0
   0.0 548.7999877929688 0.0
   0.0 548.7999877929688 559.2000122070312
   0.0 0.0 559.2000122070312
]
AttributeEnd
(Open)Inventor

- Idea: functionality first
- Support for animations
- Transformation hierarchy
- Nodes can be defined and reused
- Event nodes for interactions

```
DEF Blade Separator {  # Blade geometry and properties
  Transform {  # Blade interior
    translation 0.45 2.9 0.2
    rotation 0 1 0 0.3
  }
  Separator {
    Transform {
      scaleFactor 0.6 2.5 0.02
    }
    Material {
      diffuseColor 0.5 0.3 0.1
      transparency 0.3
    }
    Cube {
    }
  }
  Separator {  # Blade frame
    # .... (Details omitted)
  }
}
```

[The Inventor Mentor]
Virtual Reality Modeling Language (VRML)

- Similar to inventor.
- Default geometry nodes
- Complex geometry nodes use IndexedFaceSet etc.
- Trafo stack
- Materials, animations via interpolators
- portable
- Dune editor

![Dune editor]
How to define scene graphs

- Many tools use GUls for defining scene graphs
- There are **external domain-specific languages** for defining scene graphs
  - VRML
  - X3d
  - Renderman scene description
- Alternatively, there are **internal domain-specific languages**
  - Scene graph API or library, exposed by rendering lib
  - Can be written and transformed by general purpose programming language
    - Most flexible specification technique. When serializing the internal structure we (might) arrive at an external domain-specific language.
Goals of a scene graph implementation

● The core features:
  ○ Scene graph can be used to **efficiently render** the described scene
  ○ We need some mechanisms to **update scene data**

● A reusable modular system
  ○ We often want to write a scene graph library
  ○ We want an **extensible/flexible system** (can not anticipate every possible use case)
Scene graph implementation techniques

- Simple traversal based implementation should be easy, right?
- Nope
- In fact most scene graph implementations have implementation problems
- Let’s look at various implementation techniques

Any implementation ideas?
Towards a scene graph implementation

- Object-Oriented Implementation obvious.

```java
public interface Sg {
    void Render();
}
```

```java
public class Group : Sg {
    ...
    public void Render()
    {
        foreach(var c in children){c.Render();}
    }
}
```

```java
public class Renderable : Sg {
    public void Render()
    {
        GL.Draw();
    }
}
```

An Interface for scene graphs.

Implementation of node types.
Extending the object oriented approach

- Transformations can be implemented directly.

```csharp
public class Transform : Sg
{
    Sg child; Trafo t;
    public Transform(Trafo t, Sg child) {
        // ..
    }
    public void Render()
    {
        GL.PushMatrix();
        GL.MultMatrix(t);
        child.Render();
        GL.PopMatrix();
    }
}
```
Where to bind pipeline/shader inputs

- In plain OpenGL/D3D/Vulkan/GLES we know where to bind pipeline/shader inputs.
- However, this behaviour is not appropriate for a general scene graph.
  - We don’t know the shaders in advance
  - Geometry can be reused for multiple sub scene graphs
- We cannot provide input assignment for shaders.
- Thus, we need to resolve this situation when the information is present
  - Immediately before the draw call we know the shader and all its values
- The only viable solution to this is to use **semantic** identifiers.
How to define geometries

- In rendering applications there are often sophisticated mesh data-structures for specifying geometries.
- In this part of the rendering engine it is beneficial to work with flat (indexed) geometries.
- For binding geometries to arbitrary shaders, we typically want to use semantic -> Array mappings.

```csharp
class IndexedGeometry
{
    public IndexedGeometryMode Mode { get; set; }
    public Array IndexArray { get; set; }
    public Dictionary<string, Array> IndexedAttributes { get; set; }
    public Dictionary<string, object> SingleAttributes { get; set; }
}
```
The need for a traversal state

- Nodes have direct translation to graphics states? Two problems:
  - Nodes which have no direct graphics API representation
  - Nodes only the combination of which have graphics API representations -> need mechanism for communication (e.g. uniform buffers and binding locations)
- Thus: we need a place to store intermediate values.
- Solution: Equip the traversal function with a traversal state.

```java
public class TraversalState {
    public Shader Shader;...
}

public interface Sg {
    void Render(TraversalState state);
}
```
Dynamic data

- Dynamism
  - Either modify fields directly
    - Problem: we always need to track references directly into scene graph
  - Or explicitly model changeability
    - For structural changes we still need references

```java
public class Changeable<T> {
    public T Value { get; set; }
}

public class Transform2 : Sg {
    Changeable<Sg> child; Changeable<Trafo> t;
    public Transform2(Changeable<Trafo> t, Changeable<Sg> child)
    {
        // ..
    }
    public void Render()
    {
        GL.PushMatrix();
        GL.MultMatrix(t.Value);
        child.Value.Render();
        child.Value.Render();
        GL.PopMatrix();
    }
}
```
Demo

In the lecture we show the implementation of a simple scene graph system.
Common operations for scene graphs

- So far, we can render the scene graph.
- What if we would like to do other stuff like
  - Computing levels of detail
  - Writing the scene graph to disk
  - Computing the bounding box for a scene graph
- How about we add another interface member:

```java
public interface Sg3 {
    void Render(TraversalState state);
    Box3d ComputeBoundingBox(TraversalState state);
}
```
On extensibility

How to add a new node type:

- Simply add a new subclass of the interface $Sg$

How to add a new Operation:

- Simply (?) add a member to the interface

Problem:

- We want to provide a reusable library
- $Sg$ is defined in the core library
- Each user would need to edit this base interface in order to add new features !!!!
Suggestions?
The visitor pattern

```java
public interface SgVisitor {
    void Visit(Renderable2 r);
    void Visit(Group g);
    void Visit(Transform2 t);
}

public interface ISg {
    void Accept(SgVisitor visitor);
}

public class Renderable2 : ISg {
    public void Accept(SgVisitor visitor) {
        visitor.Visit(this);
    }
}

public class RenderVisitor : SgVisitor {
    public void Visit(Transform2 t) { }
    public void Visit(Group g) { }
    public void Visit(Renderable2 r) { }
}

User code can add visitors, by subclassing the SgVisitor class:

public class ComputeBoundingBoxVisitor : SgVisitor { }

You win some, you lose one:
How to add node types now?
```
The expression problem (1)

- The OOP approach:
  - Easy to add **nodes** (subclasses)
  - Hard to add new **operations** (all other nodes need to be changed)

- The Visitor approach:
  - Easy to add new **operations** (visitor implementations and subclasses)
  - Hard to add new **nodes**

- Apparently both approaches have their drawbacks
The expression problem (2)

- Formulated by Wadler in 1998 (see further reading)
  - Informally: Extensibility in both, data variants and operations while maintaining static type-safety, i.e. the compiler tells us if a node misses important implementation.
- The expression problem is a common ‘benchmark’ for programming language expressiveness
- Definition:
  - Define a datatype by cases (node types) and functions operating on them
    - Cases can be added at any time
    - Functions operating on those cases can be added at any time
  - After adding additional cases or operations, no module needs to be adapted or recompiled.
- Many non-solutions and also solutions
- Often of limited use for us :(
The expression problem in practice

- Two approaches
  - Cut back on functionality
  - Cut back on static type-safety

- Most scene graph implementations use visitors anyways.
  - e.g. OpenSceneGraph

- There are other solutions
  - Object Algebras [Oliveira and Cook 2012]
  - Look nice at first glance but hard to work with in practice.
  - Excellent paper on that topic:
    - The expression problem revisited, Torgersen 2004:
      http://www.daimi.au.dk/~madst/ecoop04/index.html
A critical view on rendering scene graphs

- Each application has its own domain “language”
  - Example: geologists use special tools in geospatial visualization (dip and strike)
- From an application developer's view, higher level of abstraction desired
  - Talk about domain entities (e.g. building, measurement tool, Flamingo, ...) instead of rendering specific entities such as renderable, trafo or shaders.
- Robert F. Toblers *Semantic Scene Graph* Implementation
  - Solves extensibility problem
  - … and provides clean separation of rendering state from conceptual state.
Research Paper 1:

Separating semantics from rendering: a scene graph based architecture for graphics applications
Analysis of the semantic scene graph approach

- Provides extensibility
  - New nodes can be implemented easily (subclass of instance)
  - New traversal can be implemented easily (subclass of traversal)
  - strictly speaking not typesafe since rule binding could fail at runtime

- Support for high-level scene description
  - via semantic scene graph

- The abstract implementation has its cost: We quickly run into performance problems!
What is the cost of scene graph traversal?

Result: Even fastest virtual call implementation can be problematic

[i7-4790, lightweight example, might be much worse in real-world scenarios]
On the performance of scene graphs…. 

- The more flexible the scene graph implementation is, the more overhead we have.
- Observation: Performance is proportional to the number of nodes visited.
- Thus, the structure/factorization of the scene graph has impact on performance.
  - This is not desirable.

- There is quite some research in the field of scene graph optimization…. 
Optimizations for scene graph systems

Common optimizations

- Reduce scene graph size
- Optimize scene graph for faster rendering

Two types

- Persistent transformations
  - Apply persistent transformation to the scene graph (commonly before runtime)
- Alternate runtime representation
  - Maintain and additional, optimized runtime representation
Example: Scene Graph Transformation

Pull up costly state changes [patent, Strauss 1999]

- Pull up and merge nodes which apply the same state.
- Find semantically equal scene graph with less nodes resulting in fewer state changes.

Assumption: A is more costly than B
Before: Set A 4 times, Set B 2 times
After optimization: Set A 3 times, change B 2 times

[Wörister 2012]
More optimizations….

- Removing redundancies
- Creating meta nodes, which apply multiple states at once
- Texture atlas creation for removing texture switches

[Wörister 2012] Removing ‘useless’ group nodes
Example: OpenSceneGraph’s optimizations

<table>
<thead>
<tr>
<th><strong>Variable</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REMOVE_REDUNDANT_NODES</strong></td>
<td>Collapse Hierarchy and Flatten Hierarchy (p. 9)</td>
</tr>
<tr>
<td><strong>MERGE_GEOMETRY</strong></td>
<td>Collapse Geometry (p. 10)</td>
</tr>
<tr>
<td><strong>TRISTRIP_GEOMETRY</strong></td>
<td>Converting geometry to triangle strips (p. 7)</td>
</tr>
<tr>
<td><strong>SHARE_DUPLICATE_STATE</strong></td>
<td>Share Attributes (p. 9)</td>
</tr>
<tr>
<td><strong>FLATTEN_STATIC_TRANSFORMS_DUPLICATING_SHARED_SUBGRAPHS</strong></td>
<td>Push transformations into vertices (p. 6)</td>
</tr>
<tr>
<td><strong>FLATTEN_STATIC_TRANSFORMS</strong></td>
<td>Same as above but without subgraph duplication</td>
</tr>
<tr>
<td><strong>SPATIALIZE_GROUPS</strong></td>
<td>Spatial Partition (p. 10) using a quadtree or octree</td>
</tr>
<tr>
<td><strong>TEXTURE_ATLASUILDER</strong></td>
<td>Generate Macro Texture (p. 10)</td>
</tr>
</tbody>
</table>

[Wörister 2012]
Geometric optimizations

- Triangle rendering vs triangle strips
- Vertex shader cache locality: Fast Triangle reordering for vertex locality and reduced overdraw [Sander et al. 2007]
- Carefully do your profiling work: papers on that topic might build on wrong assumption for your target hardware
- More on those topics later...
A critical view on persistent scene graph optimizations

- Scene graph optimizations defeat the purpose of scene graphs:
  - A clean, and understandable description of the scene
- Alternative optimization data-structures more attractive.
- Hard to implement - will see details in paper later

[Wörister 2012] Streamlined array, used at shortcut at runtime
Practical problems with optimization steps

- Most optimizations require a scene graph rewrite, or the scene graph needs to be analyzed for optimization informations.
- Observation: if we use traversal state to capture all state, the traversal state at leaf nodes contains all data we need.
- Therefore, we only need to query the leaf nodes for further use in optimization steps.
Practical problems with optimization steps

- Idea: Capture traversal states which are present at leaf nodes
- **Problem: the traversal state object mutates while traversing !!!**
- -> just capturing the traversal state variable useless
- We need to perform a deep copy of the traversal state -> expensive

General problem: optimization conflicts with dynamism....
Research Paper 2:

Lazy Incremental Computation for Efficient Scene Graph Rendering.
Resource management - approach 1

- GPU resources live in scene graph nodes directly
- Whenever we construct a rendering scene graph node (or visit it the first time), we construct the resource
Resource management - approach 1

Resources: \{Buffer_1..Buffer_4\}

To be destroyed: \{Buffer_2..Buffer_4\}
Concurrent (or batch) modification

Resources: \{\text{Buffer}_1..\text{Buffer}_4\}

To be destroyed: \{\text{Buffer}_2..\text{Buffer}_4\}

Buffer_4 is not found during \text{DisposeTraversal}
Concurrent (or batch) modification

- Need to be processed in nesting order:
  - First dispose inner nodes
- This requirement makes the thing really complex!
Collecting renderable things

Node $r$ is the root node. $s_1$ and $s_2$ are surface nodes, and $t_1$, $t_2$, and $t_3$ are transformation nodes. The geometries are represented by $g_1$, $g_2$, $g_3$, $g_4$, $g_5$, and $g_6$. Each node represents a renderable thing.

- $r$: Root
- $s_1$ and $s_2$: Surfaces
- $t_1$, $t_2$, and $t_3$: Transformations
- $g_1$, $g_2$, $g_3$, $g_4$, $g_5$, and $g_6$: Geometries
A different view

- Use notion of **renderables**
- Graphics code assembles renderables
- Renderable objects interpreted by render loop
- Render object contains all graphics state

```python
list { RenderObject_1,
    RenderObject_2,
    ...
}
```

```python
for ro in renderObjects:
    Graphics.setViewTrafo ro.Trafo
    Graphics.setShader ro.Shader
    Graphics.render ro.Geometry
```
Resource management - approach 2

- Decouple resource management from scene graph structure
- Instead of traversing the scene graph to collect resources, cache resources per renderable object
- No silver bullet, but easier in practice
Resource management

● Two options:
  ○ **The precise way:** If node is about to be removed, immediately start traversal which collects the graph's resources and destroy them
    ■ is this even possible? What if the inner state of the graph has been changed and we can’t reconstruct the original traversal?
  ○ **The lazy way:** Each time a resource is used, add it to least recently used queue. Old elements can be removed.

● Both options are problematic
  ○ Consider high-frequency switching between two variants: Here we want to be lazy
  ○ Consider high memory pressure: Often we don’t want to wait for resources to eventually go away.
From scene graphs to render objects

- There is not a single canonical scene graph structure
- Different views to structure a scene
  - Spatially
  - Semantically
  - High-performance view
- Optimizations should be carried out in separate data structures
  - Example: Scene graph and culling structure is often not the same -> Culling should be performed in a specialized geometry grid/hierarchy

- Scene graphs are fine as user API
Render objects for specific applications

- Application dictates what to include in render object
  - ‘game object’
- We focus on simpler objects which don’t capture full state, making some problems easier
- Render objects could have:
  - Transformation
  - Material
  - Mesh
- Render objects allow for optimization:
  - Game objects can be executed fast (tight loop)
  - Can be compared efficiently (good for state sorting)
Render objects in a general framework

- Very flexible representation required.
- Render objects consist of:
  - Rasterizer state
  - BlendState
  - Viewport
  - Shader
  - Uniform values
  - Vertexbuffers*
  - Indexbuffer
  - Instancebuffer*
  - Draw call description
  - ....
- What is the cost?
- Similar to Vulkan pipeline
Example: hierarchical transformations in unity

**Transform.parent**

```csharp
public Transform parent;
```

**Description**

The parent of the transform.

Changing the parent will modify the parent-relative position, scale and rotation but keep the world space position, rotation and scale the same.

See Also: [SetParent](#).
Takeaways

- Scene graphs are a common way to represent scenes
- If we want extensibility, the implementation becomes more difficult
  - Remember the *expression problem*
- The *semantic scene graph* approach provides:
  - A separation of semantic scene description and concrete graphics scene graphs
  - Rule objects can be used to capture dynamism. This way we don’t need to modify all visual states from outside of the scene graph.
Takeaways (2)

- The more flexible the implementation, the more overhead we have
  - Overheads can be significant (and performance much weaker than GPU throughput)
- Common problems of scene graph implementations (to think of)
  - Extensibility
  - Efficiency
  - Optimizations are particularly hard if traversal state can be modified arbitrarily
  - In presence of arbitrary modifications, resource management often becomes a burden
Takeaways (3)

- There are several optimization techniques for scene graphs
  - Most approaches try to reduce overheads by reducing the graph’s size
- When looking at the problem it is easy to choose the wrong path
  - It is better not to squeeze in various different ‘views’ into scene graphs!
- Better approach: compute render objects and perform optimizations on those...
- A fundamental problem arises
  - We need to constantly translate scene graphs to render objects
  - Is there a better way?
  - Could we just compute the set of render objects once, and update them accordingly to the changes?
What’s next?

- How to update render objects efficiently
- How to squeeze out performance of our graphics hardware
- How to implement high performance renderers....
Further reading

- OpenSg, A multi-thread safe foundation for scene graphs and its extension to clusters, Voß et al. 2002, [https://dl.acm.org/citation.cfm?id=569679](https://dl.acm.org/citation.cfm?id=569679)
Further Reading (2)

- The Inventor Mentor, [https://webdocs.cs.ualberta.ca/~graphics/books/mentor.pdf](https://webdocs.cs.ualberta.ca/~graphics/books/mentor.pdf)