Scene Graphs

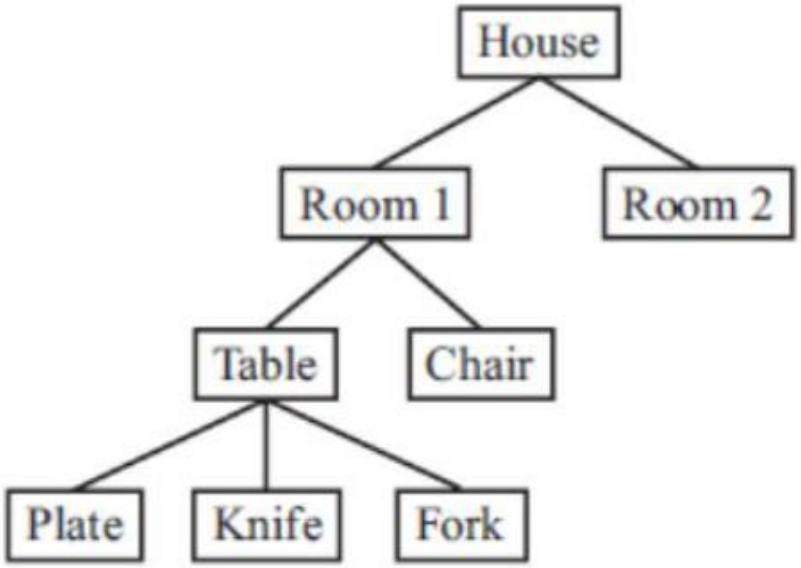
Wayne Rippin (UoD)
Dr Panagiotis Perakis (MC)

What is a Scene Graph?

- As you have seen so far, the applications you have been writing can become quite confusing quite quickly
- As soon as you have more than one object in a scene, you are mixing up updating and rendering code for the objects and it can become unclear what code is affecting which object.
- This is where a scene graph can be useful.

What is a Scene Graph?

- A game might have objects in a hierarchical relationship similar to the following:
  - The fork has a world transformation relative to the world transformation for the table
  - The table has a world transformation relative to the world transformation for the room
  - And so on...



What is a Scene Graph?

- If L_{object} *is the local transformation that* places the object in the coordinate system of its parent and W_{object} *is the world transformation* of the object, then:

$$
\begin{aligned}
W_{\text{House}} &= L_{\text{House}} \\
W_{\text{Room1}} &= W_{\text{House}}\, L_{\text{Room1}} = L_{\text{House}}\, L_{\text{Room1}} \\
W_{\text{Room2}} &= W_{\text{House}}\, L_{\text{Room2}} = L_{\text{House}}\, L_{\text{Room2}} \\
W_{\text{Table}} &= W_{\text{Room1}}\, L_{\text{Table}} = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}} \\
W_{\text{Chair}} &= W_{\text{Room1}}\, L_{\text{Chair}} = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Chair}} \\
W_{\text{Plate}} &= W_{\text{Table}}\, L_{\text{Plate}} = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}}\, L_{\text{Plate}} \\
W_{\text{Knife}} &= W_{\text{Table}}\, L_{\text{Knife}} = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}}\, L_{\text{Knife}} \\
W_{\text{Fork}} &= W_{\text{Table}}\, L_{\text{Fork}} = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}}\, L_{\text{Fork}}.
\end{aligned}
$$

A Scene Graph

- A scene graph is used to represent objects in a hierarchical relationship.
- Operations such as initialisation, updating, rendering and shutdown are recursive operations that start at the root and work their way through the tree.
- The update operation passes down the world transformation that applies at that point in the tree. Child nodes multiply their own local transformations by the world transformation passed down from their parent.

Implementation of a Scene Graph

- This is just one approach. Feel free to use an alternative approach if you wish.
- This approach makes use of the Composite *Design Pattern*.
- A design pattern is a general reusable approach to a commonly occurring problem in software design
- For more general information, see http://en.wikipedia.org/wiki/Software_design_pattern

Using the Composite Design Pattern

- See the following URL for more information: http://sourcemaking.com/design_patterns/composite
- Compose objects into tree structures to represent hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition.

Smart Pointers

- Before we look at the implementation of scene graphs, we should look at the concept of smart pointers further.
- In ‘legacy C++’, when you use pointers, you need to manage the lifetime of each object and make sure it is deleted. This can get quite difficult to manage if we are not careful, resulting in memory leaks
- Keeping track of where nodes need to be deleted can be awkward
- We end up with a lot of code that looks like this:

```
Node * nodeptr = new Node();
...
if (nodeptr != nullptr) {
    delete nodeptr;
    Nodeptr = nullptr
}
```

- This is where the use of smart pointers can be useful.

Smart Pointers

- Smart Pointers are part of the std library
- #include <memory>
- Simple RAII wrappers for raw pointers
- Two types:
  - unique_ptr: Use if an object can only have one owner at any one time
  - shared_ptr and weak_ptr: Reference counted pointers for when you may have more than one pointer pointing to an object.
- Searching on Google might lead you to examples of using auto_ptr.
- Do not use auto_ptr –this is now deprecated. It was an earlier version of what became shared_ptr.

std::unique_ptr

- Allows exactly one owner of the underlying pointer.
- Can be moved to a new owner, but not copied or shared.
- unique_ptr is small and efficient; the size is one pointer, and it supports rvalue references for fast insertion and retrieval from STL collections.
- In our case, this is not the pointer we want since as we traverse the scene-graph we could end up with a number of pointers pointing at a single node.
- However, there are times when it can be very useful.

std::shared_ptr and std::weak_ptr

- shared_ptr
  - Reference-counted smart pointer.
  - Use when you want to assign one raw pointer to multiple owners
  - The memory pointed to is not released until all shared_ptr owners have gone out of scope or have otherwise given up ownership.
  - The size is two pointers; one for the object and one for the shared control block that contains the reference count.
- weak_ptr
  - Special-case smart pointer for use in conjunction with shared_ptr.
  - A weak_ptr provides access to an object that is owned by one or more shared_ptr instances but does not participate in reference counting.
  - Required in some cases to break circular references between shared_ptr instances.
  - We will not need this for a scene graph.

Making an object pointed to by a shared_ptr

- If you used new to create an object, you would then need to pass that into the constructor for shared_ptr to create a pointer to that object.
- If you would previously have written something like:

```
Node * nodeptr;
```

- We would now replace the raw pointer nodeptr with a shared_ptr.

```
shared_ptr<Node> nodeptr;
```

Making an object pointed to by a shared_ptr

- If you would previously have written something like:

```
Node * nodeptr = new Node(param1, param2);
```

- We would now replace the raw pointer nodeptr with a shared_ptr.
- If you used new to create an object, you would then need to pass that into the constructor for shared_ptr to create a pointer to that object.
- A better way is to use make_shared.
- Using make_shared, you specify the type of the object you are creating in angle brackets and then any arguments to the constructor in brackets as normal.
- For example:

```
shared_ptr<Node> nodeptr = make_shared<Node>(param1, param2);
```

Scene Graph Construction

- This scene graph is implemented as a recursive tree structure of nodes that inherit from a base SceneNode.
- The two core nodes are SceneGraph and SceneNode.
- SceneNode nodes represent the leafs in the tree.

SceneNode Suggestion

- A suggestion for a SceneNode has been provided in SceneNode.h
- Note that no cpp file is needed since it is implemented entirely in the .h file.
- Also note that this is an abstract class –it has one or more abstract methods (i.e. no implementation), so you cannot create a new object of this type –you must create new objects of classes that inherit from SceneNode.
- For example, you can create a CubeNode class that inherits from SceneNode.

Implementation of a Scene Graph

- SceneGraph needs to be implemented

This Week

- A tutorial is provided that takes you through the steps needed to implement a SceneGraph class and use it in the DirectXFramework
- You will use this to make a robot made out of cubes.

Next Week

- You will see this week that your scene graph can end up with a lot of nodes that end up with a lot of duplicated code and data structures.
- We will look at resource management management to see how we can reduce that duplication.