CENG 242 - Homework #5

Due: 22.05.2011 23:59

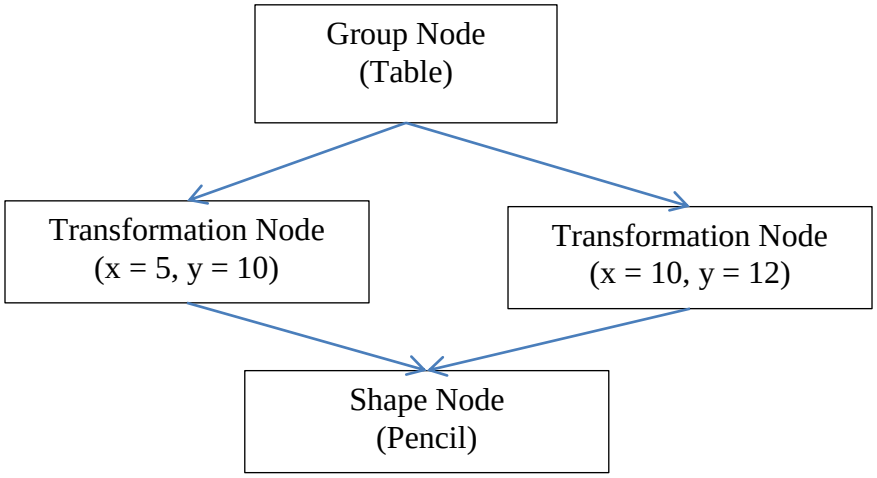
Introduction

Most computer visualization software like games, simulations or drawing software use a data structure called a Scene Graph to keep track of the things they need to draw onto the screen. You can read more about scene graphs from the Wiki page, but the basic principle is as follows: The scene is kept in a tree-like data structure, where each different type of node sends different drawing commands to the hardware. To render the whole scene, the drawing starts from the root node, and advances to its children. This is very much like traversing a tree in pre-order style. In this homework you are going to implement some classes for a very simple scene graph library. The classes you will write will be explained in the following section.

Tasks

The main difference of our scene graph from a regular tree is that, some nodes can appear in more than one place in the tree. This is like having the same identical pencil at different positions on the table. Since the pencils are identical so are the drawing commands needed to draw them. It would be a waste of memory to keep the same copy of these at every place where the pencil appears in the scene graph. Thus, we will keep just one copy, but add a reference to it at each place this object needs to be drawn at.

You can visualize this concept in the following figure. The scene graph below draws two copies of the same pencil at positions (5,10) and (10,12). Notice that the Shape Node is a child of both Transformation nodes. (A transformation node is a node that changes where its children will be drawn.)



Reference Counting

As you have seen above, a child node can have multiple parents. So the regular assumption of "delete all its children when a parent node is deleted" doesn't hold here. We need a mechanism to delete a node when it no longer has a parent, or more correctly said, when nothing refers to it. We solve this using reference counting. The main idea here is that, each node knows its reference count, and it deletes itself when that count drops to zero. The reference count is incremented each time the node is referred, and decremented each time the reference is deleted. In our homework we will do this by separating the concept of "Reference" and "Things that are Referred" in two different classes. I will begin by explaining the Reference class, but since these concepts are interdependent, you may need to look back and forth.

Reference (File: Reference.h)

```
template<class T>
class Reference {
private:
	T *obj;
public:
	// constructs this with an object pointer
	explicit Reference(T *object = 0);

	// copy constructor
	Reference(const Reference<T>& rhs);

	// destructs this Reference
	~Reference();

	// assigns another reference to this
	Reference& operator=(const Reference<T>& rhs);

	// returns the object this holds a pointer to
	T *get() const;

	// same as operator=
	void set(T* p);

	// returns the object this holds a pointer to (same as get)
	T* operator->() const;

	// returns the reference of the object
	T& operator*() const;
};
```

The first class you're going to write is called Reference. This is a template class (thus no .cpp file), and works more or less like the auto_ptr in STL. When it's given a pointer, this will be the object that the Reference class refers to. Referring to something, it will increment its reference count. That is like saying "here is one more place this object is referred". When the Reference class is destructed, for example by going out of scope, it will decrement the reference

count of the object it refers to. To increment and decrement reference counts, Reference will call addRef() and removeRef() methods of the object it points to.

Object (Files: Object.h, Object.cpp)

```
class Object {
private:
	mutable int ref_count;
protected:
	// destructor is protected. That means you can't
	// delete Objects outside of this class.
	virtual ~Object();
public:
	// construct an object. Initial ref_count should be zero
	Object();

	// increment objects reference count
	void addRef() const;

	// decrement objects reference count, and destroy
	// object when it becomes zero.
	void removeRef() const;

	// returns objects reference count
	int getRefCount() const;
};
```

Object class is the other half of the reference counting mechanism. It is the base class for all things that will be reference counted. They are always created on the heap and kept inside References most of the time. To understand how Objects and References work together to achieve reference counting, consider the program fragment below:

```
int main() {
	// Construct an object, and put its address inside a Reference.
	// When the reference is constructed with an object, it increments the
	// objects reference count to one.
	Reference<Object> a = new Object();
	// print reference count (1)
	std::cout << a->getRefCount() << std::endl;
	// When the Reference goes out of scope, it will call removeRef() of
	// the object it points to, and removeRef will detect that the object
	// is no longer needed, thus destruct itself.
	return 0;
}
```

After you finish implementing these two classes, I suggest you make sure that your implementation is correct, by testing with various cases.

Rectangle (Files: Rect.cpp, Rect.h)

```
class Rect {
public:
	int left, right, bottom, top;

	// constructs a rectangle
	Rect(int lt, int rt, int bt, int tp);

	// computes the union of two rectangles the
	// sum is a rectangle which contains both rects.
	Rect operator+(const Rect& rhs) const;

	// computes the union, and assigns the result to this.
	Rect& operator+=(const Rect& rhs);

	// scales the rectangle by multiplying each
	// component with the given number.
	Rect operator*(int scale) const;

	// scales the rectangle, and assigns the result to this.
	Rect& operator*=(int scale);
};
```

Since our scene graph is a simple 2D one, a rectangle class will be enough to handle most of the geometry. The rectangle class contains four numbers, left, right, bottom and top. A rectangle includes all points whose x coordinate lies in the closed interval [left, right] and y coordinate lies in the closed interval [bottom, top]. In a rectangle, left <= right and bottom <= top always holds. Most of the operators and methods in the rectangle are self-explanatory.

BaseNode (BaseNode.h, BaseNode.cpp)

All nodes in the scene graph inherit from the same abstract base class. BaseNode derives from Object, so it gets reference counting for free. Every node in the scene graph has a name, which is given at construction time. This name cannot be changed later.

BaseNode declares two pure virtual functions, namely getBounds() and getDrawableNodes(). The first one, getBounds, returns the bounding rectangle of a node. Bounding rectangle of a node is defined as the minimum rectangle that completely encloses a node and all of its children. The second one, getDrawableNodes(), returns the names of all nodes that partially, or completely intersect a given rectangle. A rectangle is said to be intersecting with another rectangle, if there exists at least one point that belongs to both of them.

When you call this method like,

```
nodeA->getDrawableNodes(nodeA->getBounds());
```

The method will return the names of all nodes that are in the sub tree of nodeA. When you call this method like,

```
nodeA->getDrawableNodes(Rect(4, 5, 1, 5));
```

It will return all the nodes, whose bounding boxes at least partially intersect the given rectangle. Note that the order of names you return from this function is not important.

```
class BaseNode : public Object {
protected:
	std::string name;
public:
	// constructs this with the name given in s.
	BaseNode(const std::string& s);

	// returns the list of to render. The list contains all
	// nodes (this node and its children) that fully reside
	// in or partially intersect the given rectangle.
	virtual std::vector<std::string> getDrawableNodes(const Rect& r) const = 0;

	// returns the smallest rectangle that contain
	// this node, and all its child nodes (if any).
	virtual Rect getBounds() const = 0;
};
```

ShapeNode (ShapeNode.h, ShapeNode.cpp)

ShapeNodes are the leaf nodes in the scene graph. They represent a single rectangle to be drawn on the screen. The bounds of this rectangle is set using setBounds. This is also the rectangle to be returned from the getBounds() function. ShapeNodes cannot have children, therefore the only name to be returned from getDrawableNodes is its own name if it intersects with r.

```
class ShapeNode : public BaseNode {
protected:
	Rect bnds;
public:
	// constructs a shape node with name s
	ShapeNode(const std::string& s);

	// sets the bounds of this shape node
	void setBounds(const Rect& b);

	virtual std::vector<std::string> getDrawableNodes(const Rect& r) const;

	virtual Rect getBounds() const;
};
```

GroupNode (GroupNode.h, GroupNode.cpp)

GroupNode is used to group some child nodes with a common parent. GroupNode holds references to its children. Nodes can be added and removed to a GroupNode at any time.

The bounding box of a GroupNode is the smallest rectangle that encloses all the child nodes.

```
class GroupNode : public BaseNode {
protected:
	std::vector< Reference<BaseNode> > children;
public:
	// constructs a group node with name s
	GroupNode(const std::string& s);

	// adds to the child list, and returns its
	// index in the list
	int addChild(BaseNode *child);

	// removes the child at index
	void removeChild(int index);

	// returns the child at index
	BaseNode *getChild(int index) const;

	// returns the number of children
	int getNumChildren() const;

	virtual std::vector<std::string> getDrawableNodes(const Rect& r) const;

	virtual Rect getBounds() const;
};
```

TransformNode (TransformNode.h, TransformNode.cpp)

This is a node class that is used to transform its child nodes. A transform node consists of two components: a translation and a scale. To transform a point first it is multiplied with scale, then added with translation (x, y). Initially, translation is (0, 0) and scale is 1. You should consider the transform parameters (x, y, scale) when implementing its getDrawableNodes and getBounds methods.

```
class TransformNode : public GroupNode {
protected:
	int x, y, scale;
public:
	// constructs a transform node with name s
	TransformNode(const std::string& s);

	// sets the transform parameters
	void setTransform(int x, int y, int scale);

	// gets the transform parameters
	void getTransform(int& x, int& y, int& scale) const;

	virtual std::vector<std::string> getDrawableNodes(const Rect& r);

	virtual Rect getBounds() const;
};
```

Sample Code

You are not going to submit a main function. We are going to test your codes using our own main function. Here is a sample main.cpp that will look similar to the ones we will use during grading.

```
#include <vector>
#include “Reference.h”
#include “Rect.h”
#include “BaseNode.h”
#include “ShapeNode.h”
#include “GroupNode.h”
#include “TransformNode.h”

int main() {
	// create a rectangle with width 4, height 1
	Reference<ShapeNode> shape = Reference(new ShapeNode(“pencil”));
	shape->setBounds(Rect(1, 5, 1, 2));

	// create a transform that will translate its children 4 units to left.
	Reference<TransformNode> tform1 = Reference(new TransformNode(“tform1”));
	tform1->setTransform(-4, 0, 1);
	tform1->addChild(shape.get());

	// create a second transform that will scale its children 5 units to the
	// right, and 2 units to the top
	Reference<TransformNode> tform2 = Reference(new TransformNode(“tform2”));
	tform2->setTransform(5, 2, 1);
	tform2->addChild(shape.get());

	// create a group node that will hold the entire scene
	// and add transforms as children to this group
	Reference<GroupNode> group = Reference(new GroupNode(“group”));
	group->addChild(tform1.get());
	group->addChild(tform2.get());

	Rect r = group->getBounds();
	// r will be left=-3, right=10, bottom=1, top=4

	std::vector<std::string> d = group->getDrawableNodes(Rect(3, 7, 3, 7));
	// d will contain “group”, “xform2” and “pencil”

	return 0;
	// when main returns everything should be destructed. You can check these by
	// printing stuff at destructors.
}
```

Submission Policy

- All the work should be done individually.
- In evaluation, black box method will be used. So be careful about the name of methods, data structures etc.
- You will submit your code through Cow system.
- Your codes should be written in C++ and you should submit a tar file **hw5.tar**. The tar file should include: Reference.h, Object.h, Object.cpp, Rect.h, Rect.cpp, BaseNode.h, BaseNode.cpp, GroupNode.h, GroupNode.cpp, TransformNode.h, TransformNode.cpp, ShapeNode.h, ShapeNode.cpp
- You should not have **main** function or any other global function in these files.
- Be careful about implementing reference counting properly. You will lose grades from memory leaks and segmentation faults.
- You should test your codes in **inek** machines by compiling with **g++** before submitting.