

Assignment 3: Sceneview

Algorithm Due: Saturday, October 10, 3:00 PM
Help Session: Saturday, October 10, 5:00 PM, Motorola
Program Due: Friday, October 16, 11:59 PM

1 Introduction

In order to visualize a complex 3-dimensional scene, thousands of tiny triangles must be drawn to the screen. It would be senseless to require the manual placement of each one of these triangles; instead, we usually define the scene in terms of the various primitives that compose it. Even better, we allow for primitives to be grouped, and then we reference those groups as user-defined primitives.

In sceneview, you use a TA support library to parse a cs123-specific scene file format, then interface with that library to extract all of this scene information for rendering. After you've imported all of the data, you must traverse the scenegraph (That's what the data structure in which you stored the scene info is called) and make a list of objects to draw to the screen. Each object will have certain properties needed for rendering (color, for instance), and a transformation that will move vertices and normals from object space into world space for the specific object. Once you've made this flattened list, you will iterate through the list, set up the appropriate state information, and render each object to the screen. You'll be making good use of your code from shapes and camtrans in order to accomplish these tasks.

There is a lot in this document, but your final scene viewer will fit neatly within one class. **This documentation is verbose and even redundant, but not complicated.**

2 Demo

The demo for sceneview is available on the fileserver at `/course/cs123/demo/sceneview`. Your results should be identical to the results in the demo; this includes camera positioning, camera parameterization, aspect ratio, object locations, object colors, light positions, light attenuations, and light colors.

One caveat: some scenes have primitives that have similar Z-buffer values and this causes an effect called Z-fighting. The effect causes random pixels to be rendered with the color of one primitive or another in a non-deterministic manner. Don't worry if this happens and your pixels don't exactly match the demo's. As long as the primitives are in the correct place (within epsilon) then you have done it correctly.

You can find test files with the ".xml" extension in `/course/cs123/data/scenes/`. You can feed it any of the files there, but the ones designed for intersect and ray will probably be boring in sceneview.

3 Requirements

You are required to implement a program that will instantiate the TA parser, query the resulting data structures, build a scenegraph internally, then efficiently render the scene to the display using calls to OpenGL. You will probably want to use the STL (Standard Template Library) in various places within this assignment; there are places where maps and lists will make your life much easier. See the Java to C++ tutorial for more information about the STL.

4 The CS123 Scene File Format and Parser

This assignment will make more sense once you understand the basics of the scene file format. The scenes contain objects, lights, and a camera, along with certain scalar constants that are optionally specified in each scene. (These are useful to get color levels consistent across scenes with varying numbers of lights)

Much of the information in the file format will be irrelevant in sceneview, but we advise that you store it anyway since you will need to use it in a few weeks.

The actual object specification is the most complicated section of the file format. At the top level there are a number of master objects. Each master object essentially defines a new composite primitive. They can be used elsewhere, e.g. by other master objects. A master object, however, is just a special (labeled) case of a generic tree object. Each tree consists of a list of transform blocks. Each transform block contains a list of specific transformations (which are order-specific) and either a primitive, a master object **reference**, or an unlabeled subtree. These subtrees, in turn, contain a list of transforms, and this is how the file format represents the grouping hierarchies. Here is the same information in a more readable format: (a more rigorous specification is in a supplementary handout.)

- Master Object Definition
 - Master Object Name
 - Tree Data
 - * Transform block
 - (optional translate/rotate/scale/matrix)
 - (optional translate/rotate/scale/matrix)
 - ...
 - Master Object Reference **or**
Leaf Primitive (Cube/Cone/etc), **or**
Unnamed Subtree Data (with subtree transform blocks...)
 - * Transform block
 - ...
 - ...
 - ...
 - * Transform block
 - ...
 - ...
 - ...
 - * ...
 - * ...

There is a special node in the scene graph that is the root. **Note: Every scene must have one root master object. This is the top of the object scenegraph, as the name implies.** Take a look at the scenes in `/course/cs123/data/scenes/` for examples.

What follows is a description of the parser interfaces and how to use them. It can be found in hyperlinked form by going to the cs123 web site, and clicking on the link for “Documentation”. That documentation benefits from copious internal hyperlinks, and comes highly recommended once you’ve gotten the basics from this handout.

4.1 The Top-Level CS123ISceneParser Interface

The CS123ISceneParser is a C++ interface that provides an API to access the scene graph. We have also defined a number of intermediate structs that you will need to use to access the parsed data. These data types are defined in CS123SceneData.h.

Here is the abridged header for CS123ISceneParser:

```

class CS123ISceneParser {
public:
    virtual bool getCameraData(CS123SceneCameraData& data) const = 0;

    virtual int getNumLights() const = 0;
    virtual bool getLightData(const int index, CS123SceneLightData& data) const = 0;

    virtual CS123SceneNode* getRootNode() const = 0;

    virtual bool getGlobalSceneData(CS123SceneGlobalData& data) const = 0;
};

```

- `getCameraData(CS123SceneCameraData& data)`: On return data will contain this file's camera information.
- `getNumLights()`: Gets the number of lights in the scene (possibly zero)
- `getLightData(const int index, CS123SceneLightData& data)`: On return, data will contain this file's information about the `index`'th light (specified by the parameter)
- `getGlobalSceneData(CS123SceneGlobalData& data)`: On return, data will contain this file's global data. This structure is largely irrelevant in sceneview, but you will need to use it for intersect and ray.
- `getRootNode()` : Returns a pointer to the root node of the scenegraph.

The implementation of the `CS123ISceneParser` that is specific to the xml scene format we are using is `CS123XmlSceneParser`. It has the identical methods as `CS123ISceneParser` for accessing the scene data.

```

class CS123XmlSceneParser {
public:
    CS123XmlSceneParser(const std::string& filename);

    virtual bool parse();

    virtual bool getCameraData(CS123SceneCameraData& data) const;

    virtual int getNumLights() const;
    virtual bool getLightData(const int index, CS123SceneLightData& data) const;

    virtual CS123SceneNode* getRootNode() const;

    virtual bool getGlobalSceneData(CS123SceneGlobalData& data) const;
};

```

- `CS123XmlSceneParser`: The constructor takes in the scenefile as a paramter.
- `parse()`: Calling this method will parse the scene file. Returns true if the file is valid and false otherwise.

4.2 Camera Specification

The `CS123SceneCameraData` struct encapsulates all of the camera parameters. If some items are left unspecified in the scene file, reasonable defaults are automatically selected.

```

CS123SceneCameraData {
    public:
        Point3    pos;
        Vector4   up;
        Vector4   look;

        float     heightAngle;
        float     aspectRatio;
};

```

- pos: The camera position.
- up: The camera up.
- look: The camera look.
- heightAngle: The height angle (zoom angle), **specified in degrees**
- aspectRatio: Camera aspect ratio.
- There are additional fields that can be used to describe a more robust camera model which can be implemented as extra credit for future assignments.

4.3 Light Specification

The CS123SceneLightData class encapsulates all of the light information. If some items are left unspecified in the scene file, reasonable defaults are automatically selected.

```

struct CS123SceneLightData {
    int id;
    LightType type;

    CS123SceneColor color;
    Vector4 function;

    Point3 pos; //Not applicable to directional lights
    Vector4 dir; //Not applicable to point lights

    float radius; //Only applicable to spot lights
    float penumbra; //Only applicable to spot lights
    float angle; //Only applicable to spot lights

    float width, height; //Only applicable to area lights
};

```

- LightType enum: This enumeration describes the different light types used in our scenes.
- id: This is passed to GL as the light ID. GL only allows for 8 lights in a scene, and each must have its own unique ID.
- color: The color of the light (values should all be 0-1 in every channel)
- function: The fall off function for the light.
- pos: The position of the light

- dir: For spot and directional lights, this is the direction the light is pointing.
- radius: Size of spot lights
- penumbra: Penumbra for spot lights
- angle: Max light angle for spot lights
- width: Width of area light
- height: Height of area light

4.4 Tree Node Specification

The tree node represents one node of the scene graph. It contains a number of transformations and pointers to other portions of the graph.

```
struct CS123SceneNode {
    std::vector<CS123SceneTransformation*> transformations;
    std::vector<CS123ScenePrimitive*> primitives;
    std::vector<CS123SceneNode*> children;
};
```

- transformations: A vector of all the transformations at this node. They are stored in the order that they appeared in the scene file. These transformations should be applied to the primitives and subtrees of this node.
- primitives: A vector of all the primitives at this node. For the format we are currently using, the size of this vector will always be either 0 or 1 (a single node can not have multiple primitives).
- children: A vector of the children of this node.

4.5 Transformation Object Specification

The transformation encapsulates all of the data for a single transformation. Each transformation can be either a translation, a rotation, a scale, or an “arbitrary matrix.”

You can determine the type of transformation by looking at the value in type and then access the appropriate information for that type of transformation.

```
struct CS123SceneTransformation {
    TransformationType type;

    Vector4 translate;

    Vector4 scale;

    Vector4 rotate;
    float angle;

    Matrix4x4 matrix;
};
```

- TransformationType: This enumeration exists to specify whether a given transformation is a rotation, a translation, a scale, or a custom matrix.

- translate: The translation vector for the transformation. Only valid if the transformation is a translation.
- scale: The scale vector for the transformation. Only valid if the transformation is a scale.
- rotate: The axis of rotation. Only valid if the transformation is a rotation.
- angle: The rotation angle in degrees. Only valid if the transformation is a rotation.
- matrix: The matrix for the transformation. Only valid if the transformation is a custom matrix.

4.6 Primitive Object Specification

CS123ScenePrimitive contains all of the information specifying a leaf primitive. It contains the type of the primitive as well as its properties.

```
struct CS123ScenePrimitive {
    PrimitiveType type;
    string meshfile; //Only applicable to meshes
    CS123SceneMaterial material;
};
```

- type: The type of the primitive(ie: cone, sphere, mesh)
- meshfile: The name of the file for the mesh (only application to meshes)
- material: Describes the material properties of the primitive (see below)

4.7 Material Object Specification

CS123SceneMatrrial encapsulates all the information associated with the material properties for a leaf primitive. For sceneview, you only need the diffuse color. In later assignments, you will need many of these other fields, most of which involve color and texture. If some items are left blank in the scene file, reasonable defaults are chosen.

```
struct CS123SceneMaterial
{
    CS123SceneColor    cDiffuse;
    CS123SceneColor    cAmbient;
    CS123SceneColor    cRreflective;
    CS123SceneColor    cSspecular;
    CS123SceneColor    cTtransparent;
    float              shininess;
    float              ior;
    float              blend;
    CS123SceneFileMap* textureMap;
    CS123SceneFileMap* bumpMap;
};
```

- cDiffuse: This field specifies the diffuse color of the object. This is the color you need to use for the object in sceneview. You can get away with ignoring the other color values until intersect and ray.
- **etc**: The rest of these fields are not necessary for sceneview, but will be useful in intersect and ray. You would probably be wise to parse and store them anyway, as eventually you'll have to use them.

5 Other Support Code

As in the previous two assignments, you will be implementing a OpenGL canvas that subclasses from `CS123OpenGLCanvas`. In this assignment, you will be exposed to more of the OpenGL api, specifically managing lights and specifying object transformations. We will introduce the additional parts you need to use as well as review what you have already seen.

Note that the actual triangle-drawing proper must be done inside a `glBegin()/glEnd()` block as it was in shapes. **Most importantly, note that the calls below will not work from inside a `glBegin()/glEnd()` block.** You must setup the camera, the lighting, and the current object's object-to-world transformation outside of the `glBegin()/glEnd()` block. Below are the methods from `CS123OpenGLCanvas` that you will now need to use for sceneview.

```
class CS123OpenGLCanvas
{
    public:
        // Lighting Commands
        void lightOn(int light);
        void lightOff(int light);
        void lightPos(int light, float x, float y, float z);
        void lightColor(int light, float r, float g, float b);
        void lightFunc(int light, float a, float b, float c);
};
```

- `lightOn()`: This turns the specified light on. See the id field of `CS123SceneLightData`.
- `lightOff()`: This turns the specified light off. See the id field of `CS123SceneLightData`.
- `lightPos()`: This sets the position of the specified light.
- `lightColor()`: This sets the color of the specified light.
- `lightFunc()`: This sets the falloff function for the specified light.
- **See the OpenGL handout for more information about `glVertex*()`, `glNormal*()`, and `glColor*()`.** Again, be very aware of when you are and when you are not in a `glBegin()/glEnd()` block.

For your implementation of sceneview, you will need to subclass `SceneviewCanvas` and override the `loadScene()` and `drawScene()` methods. The relevant parts of `SceneCanvas` are replicated below:

```
class SceneviewCanvas : public CS123OpenGLCanvas {
    public:
        virtual void loadScene(const char* scene) = 0;
        virtual void drawScene() = 0;
};
```

- `loadScene`: This method will be called automatically from the support code when the user selects a new scene. You will need to parse the new scene and update the canvas.
- `drawScene`: This is the function where the scene is drawn. You will need to turn on the lights, set the camera and pass to OpenGL all of the objects that need to be drawn with their positions and colors.

An important thing to keep in mind:

- CS123OpenGLCanvas does not, in any way, keep state as to what has been drawn. All changes to lighting, camera, transform, etc. only affect triangles/lines that you draw *after* you make the change. Triangles/lines already drawn will be unaffected.

6 Additional Notes

Included here are a few additional notes that are of no relevance to your grade, but that you may find useful.

- Just so you know, you will be using this code extensively for the rest of the assignments this semester. Please be careful about your design, it is important. See a TA with any questions.
- The best way to test your sceneview is to make very simple scene files that isolate particular things.

7 Extra Credit

Try adding support for a torus, or an arbitrary mesh. You could also submit a good, compelling scene file of your own. (It must be substantial to get points) Super-duper extra credit would be writing your own, un-flattened scenegraph, which would allow you to do some cool stuff come modeler time. (If you are going to try this, make sure you have a flattened scenegraph working properly first!)

8 Handing In

To hand in your assignment, type `make handin` in your assignment directory. If you have any cool scene files to show us, place them in the same directory as your source code before you hand in, and document what they are in your README file.

Handout:	Thursday, October 8	
Algorithm Handin:	Saturday, October 10	3:00 PM
Help Session:	Saturday, October 10	5:00PM, Motorola
Electronic Handin:	Friday, October 16	11:59 PM