**Computer Graphics**

**Assignment 2**

**Chris Hughes**

**Date of Submission: Friday 3rd February 2017**

**Written by Tobi Ajala (STB665)**

**Index**

**C, C++, OpenGL & OpenSceneGraph**

**Developing the Scene**

**Lighting**

**Motion Functionality**

**Record & Playback Functionality**

**Finished Program**

**Comparisons with First Program**

**Conclusion**

**References**

**C, C++, OpenGL & OpenSceneGraph**

The most fundamental difference between C & C++ programming languages is that C is a subset of C++. C was originally invented between 1969 and 1973, and C++ was developed a few years later in 1979. Both C and C++ are the developmental languages used for OpenGL and OpenSceneGraph, OpenGL which use C and OpenSceneGraph which use C++. C is oriented in the procedural aspects of the language whilst C++ is oriented in the object aspects of the language, although both languages have very similar core structures. That being said, the main difference between the two languages is the idea that OpenSceneGraph and C++ are used for Object-Oriented Programming, essentially describing the programming style associated with concepts like nodes, class, inheritance, encapsulation, etc. All of which are vital concepts, that deal with node connections, recording the scene and node movement, which is important for the completion of this project.

OpenSceneGraph is was developed to scale with a set of features called Node Kits. Node Kits allow for the developer to encompass separate libraries into the main application that can be compiled with the application or during the runtime. These Node Kits allow for a variety of effects and features to be used to develop an application further, including terrain rendering, special effects framework, character and rigid body animation, and many more that can be seen in the source code of this project.

**Developing the Scene**

Considering the given robot arm scene we were asked to develop upon, I decided to maintain the theme and build a what would look like to be a robot factory will multiple robots and other common factory robots within the scene. The introduction of low, but thick surrounding walls were used to demonstrate the ability to implement walls into the craw code but still parade the inner objects. Achieved using:

//add walls at specified location

AddWalls(rootNode, osg::Vec2(-10,-10), osg::Vec2(11,28));

root->addChild(Top\_WallGeode.get());

root->addChild(Bottom\_WallGeode.get());

root->addChild(Left\_WallGeode.get());

root->addChild(Right\_WallGeode.get());

I then proceeded with the addition of multiple robot arms, followed by tackling motion functionality before I went on to develop the scene further with lighting and texture mapping features.

**Lighting**

The implementation of lighting was achieved quickly by employing the same methods as I did in the first application. Achieved using:

//set light location

osg::Vec4f lightPosition(Location, 1.0f);

//create a local light.

osg::ref\_ptr<osg::Light> sunLight = new osg::Light;

sunLight->setLightNum(1);

sunLight->setPosition(lightPosition);

sunLight->setAmbient(osg::Vec4(1.0, 1.0, 1.0, 1.0));

sunLight->setDiffuse(osg::Vec4(1.0, 1.0, 1.0, 1.0));

sunLight->setSpecular(osg::Vec4(1, 1, 1, 1));  // some examples don't have this one

sunLight->setConstantAttenuation(1.0f);

sunLight->setDirection(osg::Vec3(0.0f, 0.0f, -1.0f));

lightSource1->setLight(sunLight.get());

lightSource1->setLocalStateSetModes(osg::StateAttribute::ON);

lightSource1->setStateSetModes(\*lightSS, osg::StateAttribute::ON);

lightGroup->addChild(lightSource1.get());

//attach light to group

root->addChild(lightGroup.get());

root->addChild(lightMarkerGeode.get())

**Motion Functionality**

Managing to get the robot arm to move was accomplished through an event handler that acted as a listener for keyboard selection. Achieved by:

//key press commands to conrol the arm movement

bool myKeyboardEventHandler::handle(const osgGA::GUIEventAdapter& ea, osgGA::GUIActionAdapter& aa)

{

switch (ea.getEventType())

{ //key press commands detection

case osgGA::GUIEventAdapter::KEYDOWN:

{

switch (ea.getKey()) {

case (osgGA::GUIEventAdapter::KEY\_Down):

std::cout << "Upper Arm moving down" << std::endl;

robotInputDeviceState->uaElevation = 0.005;

robotInputDeviceState->moveRequest = true;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_Up:

std::cout << "Upper Arm moving up" << std::endl;

robotInputDeviceState->uaElevation = -0.005;

robotInputDeviceState->moveRequest = true;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_Left:

std::cout << "Robot rotating Anti-Clockwise" << std::endl;

robotInputDeviceState->direction = 0.005;

robotInputDeviceState->moveRequest = true;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_Right:

std::cout << "Robot rotating Clockwise" << std::endl;

robotInputDeviceState->direction = -0.005;

robotInputDeviceState->moveRequest = true;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_Return:

std::cout << "Lower Arm extending" << std::endl;

robotInputDeviceState->elevation = -0.005;

robotInputDeviceState->moveRequest = true;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_Shift\_R:

std::cout << "Lower Arm retracting" << std::endl;

robotInputDeviceState->elevation = 0.005;

robotInputDeviceState->moveRequest = true;

return false;

break;

default:

return false;

}

}

case(osgGA::GUIEventAdapter::KEYUP): {

switch (ea.getKey()) {

case osgGA::GUIEventAdapter::KEY\_Right:

case osgGA::GUIEventAdapter::KEY\_Left:

std::cout << "Key released" << std::endl;

robotInputDeviceState->direction = 0.0;

robotInputDeviceState->moveRequest = false;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_Up:

case osgGA::GUIEventAdapter::KEY\_Down:

std::cout << "Key released" << std::endl;

robotInputDeviceState->uaElevation = 0.0;

robotInputDeviceState->moveRequest = false;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_Return:

case osgGA::GUIEventAdapter::KEY\_Shift\_R:

std::cout << "Key released" << std::endl;

robotInputDeviceState->elevation = 0.0;

robotInputDeviceState->moveRequest = false;

return false;

break;

case osgGA::GUIEventAdapter::KEY\_R:

//cannot record while playing

if (RecordPlayback->IsPlaying()) return false;

//stop or start as required

if (!RecordPlayback->IsRecording())

RecordPlayback->Start\_Recording();

else

RecordPlayback->End\_Recording();

break;

case osgGA::GUIEventAdapter::KEY\_P:

//cannot Play while recording

if (RecordPlayback->IsRecording()) return false;

//stop or start as required

if (!RecordPlayback->IsPlaying())

RecordPlayback->Start\_Playback();

else

RecordPlayback->End\_Playback();

break;

default:

return false;

}

}

default:

return false;

}

}

**Record & Playback Functionality**

When tackling the aspect of recording and playback for this project I had to contextualize the functionality of how this would work in a theoretical manner. From the lecture material, my understand of how this would work was to implement a form of frame counter. Essentially creating a mechanism that acknowledges every time the screen is drawn or increments a value, therefore I will have a set of numbered increments. Also, having some form of data structure than can store these drawn screens and button/key that can be selected to reset the increments back to zero; but then, every time it increments that number, it also stores the current angle and location of each of the joints. The entire functionality for this feature was developed within the Recorder.cpp source file referencing the Recorder.h header file and is achieved by:

#ifndef osg\_robot\_Recorder

#define osg\_robot\_Recorder

#include <memory>

#include <list>

#include <osg\vec4>

#include <osg\matrix>

#include <iostream>

class osg\_Recorder

{

public:

void Start\_Recording();

void End\_Recording();

void Start\_Playback();

void End\_Playback();

void AddStep(std::pair<osg::Matrixf, uint8\_t>);

std::pair<osg::Matrixf, uint8\_t> GetStep();

std::pair<osg::Matrixf, uint8\_t>  \* PeekStep();

bool StepsRemain();

bool IsRecording();

bool IsPlaying();

//factory method to prevent multiple recorders

inline static osg\_Recorder & GetRecorder()

{

if (Local\_Ptr == nullptr)

Local\_Ptr.reset(new osg\_Recorder);

return \*Local\_Ptr;

}

private:

bool Recording;

bool Playing;

inline osg\_Recorder() : Recording(false), Playing(false) {};

static std::unique\_ptr<osg\_Recorder> Local\_Ptr;

std::list<std::pair<osg::Matrixf, uint8\_t>> Data\_Points;

};

#endif

#include "Recorder.h"

std::unique\_ptr<osg\_Recorder> osg\_Recorder::Local\_Ptr;

void osg\_Recorder::Start\_Recording()

{

std::cout << "Started Recording\n";

Recording = true;

//clear any existing recording

Data\_Points.clear();

}

void osg\_Recorder::End\_Recording()

{

std::cout << "Stopped Recording at - " << Data\_Points.size() << " Frames \n";

Recording = false;

}

void osg\_Recorder::Start\_Playback()

{

std::cout << "Started Playback\n";

Playing = true;

}

void osg\_Recorder::End\_Playback()

{

std::cout << "Stopped Playback - " << Data\_Points.size() << " Frames remaining\n";

Playing = false;

}

void osg\_Recorder::AddStep(std::pair<osg::Matrixf, uint8\_t> Value)

{

Data\_Points.push\_back(Value);

}

std::pair<osg::Matrixf, uint8\_t> osg\_Recorder::GetStep()

{

//prevent exception if called without data

if (!StepsRemain()) return std::make\_pair(osg::Matrixf(), -1);

//copy last datapoint

std::pair<osg::Matrixf, uint8\_t> RetVal = Data\_Points.back();

Data\_Points.pop\_back();

return RetVal;

}

std::pair<osg::Matrixf, uint8\_t>\* osg\_Recorder::PeekStep()

{

if (!StepsRemain()) return nullptr;

else return &Data\_Points.back();

}

bool osg\_Recorder::StepsRemain()

{

//constant time operation - returns if steps still remain

return (Data\_Points.size() > 0);

}

bool osg\_Recorder::IsRecording()

{

return Recording;

}

bool osg\_Recorder::IsPlaying()

{

return Playing;

}

This task for the application happened to become the most challenging aspect of this project because I understood the theory of how it should work so well, I was confident that it could be just as obvious to execute, but I was indeed very wrong.

A significant amount of time was spent attempting to implement the playback feature.

**Finished Program**

Overall, I have managed to complete the bulk of specifications required for this assignment.

Figure ? below displays the current running application.



Displayed is the multiple robot arms, that are border by a highlight blue when clicked on. The motion for these robot arms are controlled by the user with up, down, left and right keyboards press. For functionality, the key ‘r’ starts and stops the recording feature and the key ‘p’ actions playback of any keyboard press motion made. Moving and clicking the mouse allows the user to observe lighting features considering the round orange sun acting as the camera i.e. source of light.

**Comparisons with First Program**

The thing that I was most conscious of throughout the development of the code for this assignment is the feedback received about the lack of detailed commenting throughout the raw code, though there was a lot of comments. Commenting code heavily, in the original manner I did, is a habit I established over the last couple of years from collaborative projects, but it seems that I have used comments as a more functional aspect, causing the quality of the comments themselves to drop. So, within this assignment I attempted to make a pattern of writing comments as though I’m writing them for someone who is in the beginning stages of learning about C/C++, OpenGL and OpenSceneGraph and who would require an explicit and coherent explanation of what each component of the code is doing.

Clean code is a practise I always attempt to implement, therefore your description of “a little muddled, but functional” code motivated me to re-evaluate on the structure of my code. For this program/assignment I tried to solve this by developing in a skeletal form, arranging nodes together in an order that may make things slightly more straightforward to read. As for the tasks of highlighting the robot, record & playback, etc. I chose to create additional linked source files, which I didn’t in my initial project. As much as I didn’t originally like the idea, I found that it allowed me to develop clean sets of code that are easy to read and follow.

As in the first program, I maintained simplistic keyboard handling functionality and good use of materials and lighting.

**Conclusion**

This project I had very clear goals of what I wanted to achieve by the end. Some of the objectives I aimed for I achieved, and others I didn’t but I still learnt a great deal about the relationship and developmental factors involved with OpenGL and OpenSceneGraph.

My main hindrance I not accomplishing everything I set out, I believe, was the lack of available time. I took a rigorous trial an error approach to attaining the objective of this project which can be demanding of time.

Having completed the first project and produced something I was proud of; I was quick to identify the transferrable features and prioritise them to complete towards the end of the project and focus upon the new features I was required to implement. This method did not work in my favour.

In retrospect of the current project, I can say that there is a lack in visual effects once the project is run. Adding colour and lighting to some objects were complete but more progression could have occurred with texture mapping.

**References**

Krishan Jumar. 2011. What is the difference between C & C++?. [ONLINE] Available at: <http://cs-fundamentals.com/tech-interview/c/difference-between-c-and-cpp.php>. [Accessed 31 December 2016].

Author unknown. Publishing date unknown. C++ and Object Oriented Programming [ONLINE] Available at: <http://www.studytonight.com/cpp/cpp-and-oops-concepts.php>. [Accessed 31 December 2016].

Author unknown. Publishing date unknown. About OpenSceneGraph. [ONLINE] Available at: <http://www.openscenegraph.org/index.php/about/features>. [Accessed 31 December 2016].